Professional Portfolio Documentation

Third Milestone - Software Design - INSO 4117

Prof. Marko Schütz-Schmuck

Table of Contents

Notation for Changes

Symbol	Function	Example
Highlight	Added	This text is new.
Underline	Paraphrased	This text has been modified.
~~Strikethrough~~	Removed	~~This text is no longer relevant.~~

Chapter 1. Informative Part

1.1. Team

For Milestone 1, the team was organized into four functional groups. These are Research, Functionality, Design, and Documentation; plus three Managers. In this context, all team members are considered partners. No external partners have been identified for this milestone; if any arise, their roles and responsibilities will be documented. The lists are presented in alphabetical order and indicate each group's Team Leader. The Tasks column is reserved for recording completed issues.

Managers:

Student	GitHub username	Tasks
Kaysha L. Pagan López	@Kay9876	
Juan R. Rivera Rodríguez	@JRRR0912	
Julian A. Vivas Gandarillas	@julivivas	

Developers:

Student	GitHub username	Tasks
Luis J. Cruz Cruz *(Team Leader)*	@LuisJCruz	#4, #16, #45, #69, #76, #98
Yandre Cabán Torres	@YandreCaban	#16, #35, #69, #72
Carlos A. Ferrer Hayes	@CarlosxFerrer	#16, #34, #69, #75
Osvaldo F. Figueroa Canino	@Osvaldoo1414	#16, #36, #60, #62, #69, #71, #91
Ariana P. Rodríguez Méndez	@arianrodz21	#37, #69, #73

Student	GitHub username	Tasks
Jean P. I. Sánchez Félix *(Team Leader)*	@JeanSanchezFelix	#2, #15, #23, #24, #26, #38, #45, #59, #76, #92, #96, #98
Carlos E. Cabán González	@Carloscaban	#24, #32, #33
Jahdiel E. Montero Alicea	@JadStelar	#29, #30, #31, #67
Kiara N. Pérez Rivera	@kiarancole	#38, #39, #47, #48, #60, #61, #63, #99
Pedro A. Rodríguez Vélez	@PedroRodz	#13, #14, #24, #26, #28, #38, #39, #41, #49, #52, #53, #57, #82
Alexa M. Zaragoza Torres	@alexamzt	#24, #25, #26, #27, #38, #39, #50, #51, #53, #54, #55, #56, #101

Student	GitHub username	Tasks
Diego Pérez Ganradillas *(Team Leader)*	@diegoperez16	#17, #18, #21, #45, #65, #76, #78, #93, #94, #98, #103
Axel A. Orta Félix	@Axl47	#20
Carlos Pepin Delgado	@carlospepin23	#19, #21, #40, #42, #43, #44, #83, #84, #85, #86, #181, #191, #194, #195, #197, #200, #218, #228, #336, #337, #338
Ryan R. Vélez Villarrubia	@RyanVelez101	#66, #95

Student	GitHub username	Tasks
Horeb Cotto Rosado *(Team Leader)*	@horebcotto21	#6, #12, #45, #68, #76, #98
Samarys Barreiro Meléndez	@SamarysB	#7, #77
Fernando Castro Cancel	@fernan-castro	#10, #74
Abdiel Cotto Claudio	@abdielcotto	#11, #70
Naedra J. Feliciano Agostini	@Naedra	#8, #79, #97
Heribiell A. Rodríguez Cruz	@LightPolution	#9, #38, #53, #58

1.2. Current Situation, Needs, Ideas

1.2.1. Current Situation

Moving from school to a job can be tough for grads and companies. Lots of students feel like job hunting is cold and doesn't work well. They send out tons of applications and don't hear back. Forbes Advisor says almost 60% of job seekers feel like companies just disappear on them, which is annoying and makes them worried about their careers. But companies have problems, too. They get flooded with applications for starter jobs, and a lot of people aren't a good match. It takes up a lot of time and money to find the right people. This whole thing is inefficient and doesn't really show when a candidate and a company might actually be a good fit for each other's goals. It can be a pretty bad experience for everyone.

1.2.2. Need

Structural friction in early-career recruiting (lack of visibility, opaque communication, and operational overload) reveals needs that belong to domain actors, not to a platform. The needs are outlined below.

Applicant / Recent Graduates

- Be discoverable for roles that truly match skills, interests, eligibility, availability, and location.
- Receive timely, transparent feedback about interest and status to reduce ghosting and uncertainty.
- Ensure fair access to opportunities and maintain control over privacy when sharing work and

personal data.

- Obtain guidance to translate coursework, projects, and soft skills into recruiter-trusted signals.

Employers / Recruiters

- Efficient triage toward candidates who meet must-have criteria and show mutual interest.
- Rich, trustworthy evidence of capability and professional culture.
- Predictable, compliant communication and scheduling to minimize drop-off and miscommunication.

Cross-Cutting Needs

- Mutual-interest signaling before deep engagement.
- Early expectation alignment on role scope, compensation range, work modality, and timeline.
- Low-friction coordination for first conversations and follow-ups.
- Trust and safety: identity assurance, respectful conduct, and clear reporting channels.

Project-Internal Enabling Needs

- A shared domain description and a baseline set of requirements so the team understands needs independently of any system-to-be.
- A consistent, ubiquitous language across analysis, design, and code to prevent concept drift.
- Plans for requirements, architecture, component design, implementation, and testing to support whatever idea is chosen later.

1.2.3. Ideas

We propose a three-part design focused on a personalized, efficient, and high-quality user experience. The foundation of this approach is a onboarding and profile system. The system would create two fundamentally different experiences based on the user, whether they are a recruiter or a candidate. The system will request only the most relevant information for each persona, such as portfolios for students or verifying company details for recruiters. The system will have an interface that avoids clutter and ensures the platform feels built for each user from their very first interaction. Making it easier and more inclusive without replacing the current infrastructure.

Once users are onboarded, the swiping mechanism would enhance the core matching process by moving beyond a simple binary decision. This means creating carefully designed cards that act as a information display. The profiles can have an simple view and a more detailed view. The key to this design is a hierarchy that is informed by user research and which surfaces key decision making data relevant to the user directly in the swiping interface to maximize informed matches without causing overload.

Finally, to ensure connections are meaningful and productive, the mutual Match connection and messaging gateway would unlock only after both parties have shown interest. Afterwards, the system would immediately facilitate the first message and it could include some kind of icebreaker or customizable openers. Furthermore, a dedicated inbox to keep users organized, allow for easy

profile review, and potentially integrate scheduling tools, transforming a simple Match into a genuine gateway to opportunity.

1.3. Scope, Span, and Synopsis

1.3.1. Scope and Span

Scope

The project's scope is to develop a mobile application aimed at improving the connection between students and recruiters. The app will address issues with traditional job search platforms and career fairs, which are often impersonal and inefficient, leading to a lack of engagement and missed opportunities. The project will encompass several key areas:

- Domain Engineering: Analyze the current landscape of student-recruiter engagement, identifying pain points in job fairs, static job boards, and passive search platforms. The goal is to create a faster, more efficient, and more engaging way for students and recruiters to connect.
- Requirements Engineering: Define system requirements to enable students to showcase their skills, qualifications, portfolios, and preferences dynamically. Recruiters will also be able to display what their company is offering and looking for. Requirements will focus on improving job placement rates, event attendance, and reducing the time spent in the recruitment process. These requirements will be refined continuously using direct feedback from both students and recruiters.
- Software Architecture: The architecture will feature a mobile front-end with a swipe-based matching system, real-time notifications, and event integration. The back-end will connect with job boards, applicant tracking systems, career services, and on-campus event data to strengthen student-recruiter engagement.
- Software Design Process: The project will follow an iterative design and development process, beginning with a pilot test to evaluate performance and identify areas of improvement. User feedback will drive optimization of the user interface, swiping experience, and the matching algorithm.

Span

The project's span is focused on creating a scalable and user-friendly solution that streamlines the student-recruiter connection process. The app is designed to support efficient matching, real-time communication, and event integration.

- Specifics of the System: Students can create detailed profiles including videos, portfolios, and soft skills. Recruiters will also create company profiles that highlight roles, culture, and expectations. When both parties swipe right, they are notified of a Match and can begin communicating via chat or set up interviews. The app will also notify students about campus events that involve companies they have swiped right on, even if a Match has not occurred.
- Target Audience and Expansion: The initial span of the project involves a pilot test with a defined user base of students and recruiters. Expansion will include partnerships with recruiters, direct marketing to universities, and support for on-campus career fairs. Over time, the platform will expand to larger student and recruiter networks beyond the initial pilot.

- Methodology and Maintenance: The project will adopt an iterative methodology with regular update cycles guided by new technology trends and continuous user feedback. Effectiveness will be tracked through key metrics such as app usage frequency, Match success rate, recruiter follow-up rate, event attendance, and user satisfaction. The cycle of feedback, optimization and scaling will ensure the app remains relevant and impactful.

1.3.2. Synopsis

The project aims to develop a mobile application that modernizes how students and recruiters connect by addressing the inefficiencies of traditional job search methods. It uses a swiping-based interface to enable dynamic and real time engagement between students presenting their qualifications and recruiters offering opportunities. Throughout its lifecycle, the project will progress through several key phases: requirements engineering to define user and system needs, software architecture and design to establish the platform's structure, implementation of both front-end and back-end components, and systematic testing and validation to ensure reliability and usability.# #An iterative approach will be adopted, allowing feedback to refine requirements and improve design over time. The final goal is to deliver a scalable and efficient application that enhances job placement success, fosters meaningful recruiter to student connections, and maintains user centered quality across all development stages.

1.4. Other Activities than Just Developing Source Code

Although implementation is central to the project, a significant portion of the work since Milestone 1 consisted of non-coding activities that shaped the system's direction, clarified requirements, validated assumptions, and prepared the team for upcoming development. This section documents planning, research, experimentation, testing preparation, and design exploration work that contributed directly to the system's maturity.

1.4.1. Requirements Refinement and Domain Understanding

Throughout Milestones 1-3, the team conducted several rounds of requirement clarification to reinforce the foundation of the system and align with the evolving domain model (Candidate, Recruiter, Profile, Opening, Application).

Domain Exploration Workshops

- Structured sessions focused on understanding student–recruiter interactions in job fair environments.
- Observations emphasized real-world constraints such as noise, fatigue, time limitations, lack of preparation, and poor recall.
- Insights from these workshops directly influenced the definition of aggregates and bounded contexts.

Persona and Scenario Development

- Personas included: overwhelmed student, fatigued recruiter, early-year undecided student, and highly prepared senior.

- Scenarios were refined to represent realistic workflows and stress points across both user types.

Requirements Revisions

- Several ambiguous requirements were rewritten to be measurable and testable.
- Non-functional requirements (responsiveness, clarity, adaptive interaction, information recall support) were clarified.
- Edge-case analysis led to new requirements around profile completeness and recruiter note structure.

Mapping Requirements to Domain Structures

- Requirements were systematically mapped to the emerging aggregates:
  - **Candidate** – profile ownership, application submission, match visibility.
  - **Recruiter** – opening ownership, candidate review workflows, note-taking.
  - **Profile** – central reference for candidate attributes and matching.
  - **Opening** – job description structure, eligibility constraints.
  - **Application** – lifecycle events, status handling, duplicate prevention.

These activities ensured a deeper understanding of user needs and established a solid basis for design and implementation.

1.4.2. Design and Architecture Exploration

Multiple design-oriented sessions occurred to visualize and reason about the evolving system structure.

Domain Model Iterations

- Iterative refinement of early conceptual models revealed issues such as duplicated responsibilities in the Profile entity and unclear aggregate boundaries.
- Responsibilities were redistributed to reduce coupling, especially between Candidate and Recruiter roles.

UML Diagram Creation

- Class diagrams were drafted for all primary aggregates (Candidate, Recruiter, Profile, Opening, Application).
- Sequence diagrams were created for:
  - Candidate applying to an opening.
  - Recruiter reviewing matches.
  - MatchService generating a ranking.
- State diagrams were explored for application status transitions.

Architecture Discussions

- Evaluations included modular monolith vs. service-split designs.
- Clearer separation between application layer and domain layer emerged from these discussions.
- The team emphasized a domain-driven design approach for maintainability and scalability.

Review of Aggregates and Bounded Contexts

- Candidate and Recruiter emerged as primary user-driven aggregates.
- Profile, Opening, and Application were validated as supporting aggregates with well-defined ownership.
- Ambiguous relationships (e.g., who owns profile edits during applications) were clarified.

Preliminary Interface and Interaction Flow Mapping

- Early workflow mapping included:
  - Profile creation.
  - Opening creation.
  - Application submission.
  - Match generation.
  - Recruiter feedback loop.

These activities informed structural and behavioral foundations for upcoming implementation.

1.4.3. Research and Technology Evaluation

Significant research effort took place to guide development decisions and ensure alignment with industry standards.

Data Storage Approaches

- Comparison of relational vs. document-based models for profile and recruiter note data.
- Early sketches of hybrid approaches where structured attributes coexist with semi-structured notes.

Review of Similar Existing Systems

- Analysis of Handshake, LinkedIn, and other career platforms identified common features and missing opportunities.
- Findings informed the matching and interaction design.

Matching Algorithm Research

- Early exploration of:
  - attribute weighting,

    - recruiter preference scoring,
    - profile completeness coefficients,
    - potential future ML extension.

This research shaped the conceptual basis for MatchService and related components.

1.4.4. Testing Activities

This involves deciding how unit testing, integration testing, and usability evaluation will eventually be carried out once prototypes and code are available. In parallel, version control practices such as branching strategies, pull requests, and code reviews can already be defined so the team is prepared to manage collaboration effectively when development begins. These preparatory activities set the standard for a structured and reliable workflow.

Early Validation and Experimentation

- Manual walkthroughs simulating student–recruiter interactions.
- Scenario-driven tests examining typical flows:
    - match generation,
    - profile editing,
    - application duplication.
- Pseudocode logic experiments for validating data flow in Profile and Opening interactions.
- Persona-based consistency checks to ensure realism.

Planning and Infrastructure for Future Testing

- Draft test strategy including unit tests, integration tests, and future usability tests.
- Early test case definitions:
    - Profile completeness,
    - Application submission rules,
    - Opening eligibility checks.
- Error scenario mapping for likely failure points.

These activities ensured analytical rigor before any code was written.

1.4.5. Project Management and Coordination

Consistent coordination activities supported progress clarity and team alignment.

Milestone Planning Sessions

- Deliverables, deadlines, and responsibility distribution were set for each milestone.

Meeting Documentation

- Logs captured decisions, unresolved questions, design changes, and next steps.

Role Clarification

- Clear assignments for:
  - domain modeling,
  - testing preparation,
  - documentation,
  - research.

Communication Management

Project management and communication establish the general structure that ties everything together during this milestone. Setting clear milestone goals, assigning responsibilities, and documenting meeting outcomes help the team stay organized and avoid confusion. Regular communication ensures that issues are identified and addressed early, while planning for security, privacy, and future phases prepares the project for ongoing development. By combining documentation, early planning for testing and version control, and strong management practices with the coding that will follow, the team lays the foundation for a successful project.

Documentation and Logbook Updates

Source code development is a top priority on this project, its success depends on several activities that extend beyond programming. Documentation plays a crucial role in keeping the project aligned, covering goals, requirements, architectural decisions, and detailed contributions. Well-maintained records make it easier for members to integrate into new teams and ensure stakeholders remain informed about progress, scope, and direction throughout the project.

Domain Description Drafting

- Consolidation of observations, stakeholder statements, personas, and requirements.

Milestone Logbook Maintenance

- All design and modeling changes recorded with proper notation and revision trails.

Diagram Documentation

- All UML diagrams versioned with explanations for each update.

1.5. Derived Goals

A key outcome of this project is the emergence of several derived goals that extend beyond the primary function of matching recruiters and job seekers. These goals leverage the platform's unique data and position to create secondary, significant value.

One major derived goal is the generation of actionable market intelligence. The matching process naturally produces a rich stream of data on skills, salaries, and hiring trends. By systematically learning from this data, this intelligence enables strategic talent acquisition for recruiters by identifying hidden talent pools, providing competitive benchmarking on metrics like time-to-hire, and predicting emerging skill demands. Furthermore, this aggregate data serves as a critical feedback loop for the platform's own strategic development and product roadmap, ensuring the service remains optimally aligned with real-world market needs. This transforms the platform from a simple transactional tool into an indispensable, intelligent partner for talent leaders and the platform itself.

A second derived goal involves creating a dynamic bridge between education and industry. The data generated provides a real-time, aggregate map of the skills employers need versus the skills the collective candidate pool possesses. This allows the platform to become a vital feedback mechanism for universities and bootcamps, helping them validate and modernize their curricula to close specific, large-scale skills gaps, such as a lack of training in high-demand tools like Docker. On a broader scale, this data can fuel regional economic development by helping cities and governments identify critical talent shortages and create targeted workforce programs, ultimately building a stronger, more aligned local economy.

Finally, a crucial derived goal is to transform the job search from a black box into a guided journey for the candidate. The platform can leverage its collective data to demystify the process, providing unprecedented transparency. This includes empowering job seekers with data-backed salary insights that illuminate how specific skills impact pay, enabling confident negotiation. It also involves providing a realistic "mirror" to the market, showing candidates exactly how their profile measures up and pinpointing specific areas for improvement. By offering a "pulse" on typical hiring timelines and illuminating common career progression paths, the platform reduces the anxiety and uncertainty of job hunting. This shifts its value proposition from merely finding a job to becoming a trusted advisor for managing an entire career, thereby building immense user loyalty and trust.

Chapter 2. Descriptive Part

2.1. Domain Description

2.1.1. Domain Rough Sketch

NOTE	This is an unprocessed collection of notes, quotes, and observations from the domain (student–recruiter interactions). However, analytic abstractions derived from these notes have been added inline.

Student (Abdul, Mechanical Engineering senior): “At last year’s fall job fair, I stood in line almost 40 minutes just to hand my résumé to a recruiter from Pratt & Whitney. The line wrapped around the Student Center atrium — people were sweating, holding folders, and trying to look composed. When I finally got to the table, the recruiter just smiled, took my résumé, and said they’d be in touch. I wasn’t sure if she’d even remember me by the end of the day.”

- → **Observation:** Waiting fatigue, impersonal exchanges, uncertainty about follow-up.
- → **Derived concepts**: *queue duration*, *interaction brevity*, *candidate recognizability*, *physical discomfort*, *interaction outcome uncertainty*.

Recruiter (Corey, Lilly Pharmaceuticals): “We meet hundreds of students in just a few hours. By the end of the day, names and faces blur together. I jot quick notes on sticky labels: ‘Python — confident’, ‘Good communicator’, or sometimes just a checkmark. Later, when reviewing, I can’t always recall which student said what.”

- → **Observation:** Recruiters rely on quick, shorthand impressions; limited retention of individual details.
- → **Derived concepts**: *meeting duration*, *candidate trait extraction*, *note/annotation*, *memory fade*, *reviewing constraints*.

Student (Maria, Computer Engineering junior): “This year I wore new shoes — bad idea. I had a class right before the fair and another right after, so I kept rushing back and forth between buildings. By the time I came back to the fair after my thermodynamics class, it was packed. I could barely hear anything; someone even stepped on my heel while I tried to talk to a recruiter from Google. It’s so loud you have to almost shout to be heard.”

- → **Observation:** Physical fatigue and environmental noise affect both comfort and communication quality.
- → **Derived concepts**: *environmental noise*, *communication friction*, *mobility constraints*, *time pressure*, *crowd density*.

Recruiter (Daniel, local tech startup): “By noon, my voice is gone. I try to smile and look engaged, but it’s hard to remember each student’s story. We keep a spreadsheet open on the booth laptop — name, major, key skills — just enough to remember who’s who. If a student brings something visual, like a project or a QR code to their portfolio, it really helps.”

- → **Observation:** Recruiters value portfolio evidence over résumé lines; digital aids improve

recall.

- → **Derived concepts**: *information reinforcement*, *portfolio artifact*, *digital reference*, *interaction fatigue*, *memory support tools*.

Student (Lina, Electrical Engineering sophomore): "I applied on Handshake, then on LinkedIn, and again on the company website. I never knew if anyone actually saw it. After two months, no one replied, and I started to wonder if my applications were just getting lost somewhere."

- → **Observation:** Lack of transparency in the post-fair pipeline; unclear feedback loop between applicant systems.
- → **Derived concepts**: *application visibility*, *pipeline opacity*, *multi-platform submissions*, *response latency*.

Recruiter (Rafael, mid-sized design firm): "We usually contact students weeks later, but by then many already accepted other offers or stopped checking their student email. If the first batch of candidates looks good, we stop looking. There's just too little time."

- → **Observation:** Recruiting process favors early applicants; short attention window post-fair.
- → **Derived concepts**: *review window*, *candidate availability*, *early-selection bias*, *time-limited screening*.

Student (Jorge, freshman): "I was nervous to talk to recruiters because I didn't even know what their companies did. Some just told me, 'come back next year when you have more experience.' I mostly went this year just to practice — to get used to talking."

- → **Observation:** Early-year students attend fairs for exposure rather than placement; recruiters focus on upperclassmen.
- → **Derived concepts**: *experience gap*, *interaction purpose variance*, *preparation asymmetry*, *first-time candidate behavior*.

Recruiter (Helen, aerospace company): "A lot of students come unprepared — they don't know which positions are open, or what our company even does. We'd love to pre-screen them, maybe through a short survey or a skill-based filter before the fair starts."

- → **Observation:** Recruiters express need for pre-fair filtering and digital preparation workflows.
- → **Derived concepts**: *preparedness level*, *pre-screening questionnaire*, *role-awareness*, *skill matching*.

Student (Emilio, Software Engineering senior): "The fair feels like speed dating — you only get two minutes to impress, then they move on. It's hard to show personality or teamwork skills when the line behind you is ten people long."

- → **Observation:** Limited interaction window leads to superficial evaluation; "soft skills" remain under-assessed.
- → **Derived concepts**: *interaction time constraint*, *soft-skill invisibility*, *queue pressure*, *rapid-assessment mode*.

Recruiter (Isabel, consulting firm): “We hand out branded tote bags and pens — it’s funny, but sometimes that’s what students remember us by. When I reach out later, they’ll reply ‘Oh, you were the ones with the blue water bottles!’”

- → **Observation:** Brand association through physical tokens enhances recall; small sensory cues matter.
- → **Derived concepts**: *brand token*, *sensory cue*, *recall aid*, *brand-to-memory mapping*.

Student (Arjun, Industrial Engineering senior): “I wish I knew right away if I had a chance instead of waiting months. After the fair, you check your email every day, but mostly nothing happens.”

- → **Observation:** Desire for instant or early feedback mechanisms post-interaction.
- → **Derived concepts**: *feedback latency*, *candidate anxiety cycle*, *instant-evaluation desire*.

Recruiter (Paula, tech recruiter): “We often rehire interns we already know. There’s trust there — we’ve seen their work. That’s why sometimes new applicants don’t get the same attention.”

- → **Observation:** Prior familiarity skews fairness; existing relationships heavily influence hiring behavior.
- → **Derived concepts**: *familiarity bias*, *trust-weighted selection*, *returning applicant advantage*.

Student (Elena, Business Administration major): “The fairs are always packed — the noise, the pushing, it’s chaotic. I tried to ask a recruiter a question, but they just handed me a flyer and moved on. Later, I realized that the quieter booths in the back actually had time to talk.”

- → **Observation:** Spatial arrangement and booth placement affect engagement quality.
- → **Derived concepts**: *booth location*, *interaction density*, *engagement opportunity*, *crowding effect*.

Career Advisor (Lucia, UPRM): “We encourage students to bring a ‘target list’ of companies, but many still wander without a plan. On the other side, recruiters tell us they want better visibility into who’s coming — maybe an advance roster of students and their profiles.”

- → **Observation:** Advisors recognize preparation asymmetry; opportunities for platform pre-matching.
- → **Derived concepts**: *target-list preparation*, *candidate pre-registration*, *roster visibility*, *pre-match potential*.

2.1.2. Terminology

NOTE

Each term below is derived from raw observations in the Domain Rough Sketch (2.1.1) and refined through concept analysis. Terms are presented with their classification (entity, actor, event, etc.), domain scope, and traceability to source observations. The derivation of each term is explicitly shown through annotations linking back to specific observations, quotes, and patterns from the rough sketch.

This structured approach ensures transparency in how our domain vocabulary

> emerged from real-world observations rather than being imposed artificially. When new terms are needed, they should follow this same pattern of clear derivation from documented domain phenomena.
>
> The terms are now organized into Domain Nouns, Domain Verbs, and Domain Constraints to align with ubiquitous language structure in Domain-Driven Design.

Domain Nouns (entities, objects, abstractions)

Applicant

(entity, domain) A person, usually a student or recent graduate, pursuing professional chances. Candidates strive to highlight their abilities and credentials via their portfolios. *Derived from: "Students often rely on school provided career services for resume templates"; "Student: 'I applied through Handshake, LinkedIn, and the company website'"*

Employer

(entity, domain) An organization that owns openings and ultimately employs candidates. Multiple recruiters may represent the same employer during sourcing and selection. *Derived from: "Recruiter: 'We usually contact students weeks later'"; multiple references to company and employer context*

Recruiter

(actor, domain) A hiring professional acting on behalf of an employer to discover, evaluate, and engage candidates. A candidate may interact with several recruiters for the same employer. *Derived from: "We meet hundreds of students in a single afternoon"; "Some recruiters only target juniors and seniors"; "Recruiters: 'We prefer quick ways to identify students with the right skills'"*

Portfolio

(entity, domain) A collection of an applicant's work, projects, and achievements. Portfolios provide recruiters with evidence of professional skills.

Skill

(entity, domain) A demonstrated ability, either technical or interpersonal, that contributes to an applicant's professional profile.

Qualification

(entity, domain) An educational or professional credential (e.g., degree, certification) that indicates formal preparation or eligibility.

Work Modality

(entity, domain) The way in which work is performed, such as on site, remote, or hybrid. *Derived from: Students value flexibility in work arrangements; need for early expectation alignment on work modality*

Compensation Range

(attribute, domain) The expected or offered salary **plus applicable benefits** (e.g., equity/options, health plan, stipends, shuttle/transport, on-site meals). Considered between applicant and employer. *Derived from: "Early expectation alignment on role scope, compensation range, work modality, and timeline"; need for transparency in total compensation*

Location

(attribute, domain) The geographic context for a role or event (e.g., city/region/country) or “remote-eligible,” used for discovery and filtering.

Start Date

(attribute, domain) The intended employment start date or window associated with an opening or offer.

Mutual Interest Signaling

(event, domain) The occurrence of both applicant and employer expressing interest, creating the basis for a potential connection.

Connection

(entity, domain) The relationship established upon mutual interest. Day-to-day interaction typically occurs between the applicant and the **recruiter** representing the **employer**; any employment outcome is with the employer.

First Conversation

(event, domain) The initial professional interaction after a confirmed connection, typically between an applicant and a **recruiter**; it may still be considered as leading to a connection with the **employer**. *Derived from: "Student: 'I’m nervous approaching a recruiter if I don’t already know about the company'"; importance of structured initial interactions*

Interview Modality

(taxonomy, domain) The manner in which an interview is conducted (e.g., in-person, virtual). Serves as the parent concept for specific interview types. *Derived from: "Job fairs are often loud, crowded, chaotic"; need for flexible interaction formats*

Validity Period

(attribute, domain) The time window during which an offer remains actionable before it expires.

~~**Clarifications**~~

~~(process, domain) Bidirectional questions and answers to resolve ambiguities (scope, duties, timeline) without changing negotiated terms.~~

~~**Adjustments**~~

~~(process, domain) Mutually agreed changes to offer terms (e.g., title, start date, compensation range) prior to acceptance.~~

~~**Accept**~~

~~(decision/event, domain) The applicant’s affirmative decision to proceed under the current offer within its validity period.~~

~~**Decline**~~

~~(decision/event, domain) The applicant’s explicit decision not to proceed under the current offer.~~

~~**Ghosting**~~

~~(behavior, domain) The act of ceasing communication without notice, leading to inefficiency in~~

~~the recruitment process.~~ ~~*Derived from: "Student: 'I never know if recruiters actually looked at my resume or if it went into a pile'"; lack of feedback and communication*~~

Identity Assurance

(behavior, domain) The process of verifying that participants are authentic and represent legitimate individuals. *Derived from: Need for "trust factor" mentioned by recruiters; importance of verified connections*

Recruitment Event

(entity, domain) A scheduled occasion, such as a job fair or networking session, where applicants and employers directly engage. *Derived from: Multiple references to job fairs; "At the job fair, I stood in line 40 minutes just to hand over my resume"*

Expectation Alignment

(behavior, domain) The process of clarifying and agreeing on key role aspects, including scope, compensation, timeline, and modality. *Derived from: "Recruiters say a lot of students come unprepared, don't know what positions are open"; need for upfront clarity*

Trust and Safety

(behavior, domain) The assurance that professional interactions occur under respectful conduct, secure data handling, and clear reporting mechanisms. *Derived from: Recruiters mentioning "trust factor" with known candidates; need for safe, professional interactions*

~~**Feedback**~~

~~(event, domain) Information shared between employer and applicant regarding application status, interest, or evaluation, enabling transparency.~~ ~~*Derived from: "Student: 'I wish I knew immediately if I had a chance instead of waiting months'"; need for timely updates*~~

User

(technical/authentication, domain) An authenticated account in the system. Each User is typed as either Candidate (e.g., Student) or Recruiter; avoid using "User" to describe domain roles.

Student

(subset, domain) A Candidate currently enrolled at a university/college. Used when context involves campus events, coursework, or student services.

Profile

(entity, domain) The core representation of a User in the system (typed as StudentProfile or RecruiterProfile). Distinct from a Profile Card used for swiping.

StudentProfile

(typed entity, domain) A Candidate's profile containing résumé, skills, preferences, portfolio items, and visibility settings. Identified by an immutable UUID.

RecruiterProfile

(typed entity, domain) A Recruiter's profile including employer association, role/title, sectors, location, and verification status.

Profile Card

(ui artifact, domain) Condensed, swipeable representation of a Profile shown in the Discovery Feed. *Derived from: "Recruiters: 'We prefer quick ways to identify students with the right skills'"; need for efficient profile scanning*

Discovery Feed

(experience, domain) A personalized deck of Profile Cards presented for evaluation. *Derived from: "After a while, names and faces blur together"; need for structured, paced discovery*

Swipe

(action, domain) The primary gesture to evaluate a Profile Card. Right-swipe = Like; left-swipe = Pass. *Derived from: "Students compare the process to 'speed dating'"; need for quick, clear interest signals*

Like

(action, domain) An expression of interest on a Profile Card (right-swipe). Stored by the system for Match evaluation.

Pass

(action, domain) A dismissal on a Profile Card (left-swipe). Removes the card from the current session.

Match

(event, domain) Created only when both sides have explicitly liked each other's Profile Cards (mutual interest signaling).

~~**Message**~~

~~(entity/action, domain) A communication exchanged only when a valid Connection exists (or explicit permission).~~ ~~*Derived from: "Informal hallway conversations sometimes lead to opportunities"; need for structured yet natural communication*~~

Opening (Job Opening)

(entity, domain) A role posted by an Employer with explicit Requirements, Location, Work Modality, Compensation Range, and Start Date. *Derived from: "Recruiters say a lot of students come unprepared, don't know what positions are open"; need for clear role definition*

Requirements

(structure, domain) Must-have and nice-to-have criteria for an Opening (e.g., skills, eligibility, language, authorization). Used to assess Eligibility. *Derived from: Recruiter notes like "Has Python", "Strong communication", "Not ready"; need for structured evaluation criteria*

Eligibility

(assessment, domain) Whether a Candidate meets the Requirements of an Opening (meets / partially meets / does not meet).

Shortlist

(collection, domain) A curated set of Candidates selected by a Recruiter for next steps (review, outreach, interview).

Interview

(event, domain) A scheduled conversation following a Connection/Shortlist; must respect non-overlapping time blocks and uses an Interview Modality.

~~**RSVP**~~

~~(action/state, domain) An explicit intent to attend an Event; updates capacity and powers reminders.~~

Offer

(entity, domain) A proposal from an Employer to a Candidate with explicit terms (role, Compensation Range, Location/Work Modality, Start Date) and a Validity Period.

Notification

(system event, domain) An in-app alert for key events (e.g., Match created, unread Message, Event reminder, Offer updates).

Visibility

(setting, domain) Exposure level of a StudentProfile: Public (searchable), By Match (visible only to the matched party), or Private (not discoverable; shared explicitly).

Session

(technical, domain) The authenticated runtime context for a User. Authorizes actions (swipes, messages, RSVPs).

Domain Verbs (actions, events, operations)

Apply

(action, domain) A student submits an Application to an Opening, freezing artifacts at submission time. *Performed by: Student | Purpose: Enables triage and interview selection based on consistent, versioned evidence.*

Publish

(action, domain) Make a Profile or Company page visible according to Visibility settings. *Performed by: Student / Recruiter | Purpose: Allows discovery and evaluation within the matching process.*

Filter

(action, domain) Narrow candidates or openings by skill, modality, authorization, or availability. *Performed by: Recruiter / System | Purpose: Supports fast triage and prioritization of relevant matches.*

Swipe

(action, domain) Evaluate a Profile Card using a Like or Pass decision. *Performed by: Student / Recruiter | Purpose: Generates interest signals used to determine Match eligibility.*

Like / **Pass**

(action, domain) Rapid interest-signaling gestures that drive Match formation. *Performed by: Student / Recruiter | Purpose: Determines if a Match will be created.*

Match

(system event, domain) Created when both sides explicitly Like each other's Profile Card. *Performed by: System | Purpose: Establishes a Connection and unlocks communication.*

Clarify

(process, domain) Exchange bidirectional questions and answers to resolve uncertainties without altering terms. *Performed by: Student / Recruiter | Purpose: Reduces ambiguity and supports informed decision-making.*

Adjust

(process, domain) Modify terms such as title, start date, or compensation before acceptance. *Performed by: Student / Recruiter | Purpose: Supports negotiation and alignment before final decision.*

Message

(action/event, domain) Exchange communication once a valid Connection exists. *Performed by: Student / Recruiter | Purpose: Coordinates next steps and reduces delays or ghosting.*

RSVP

(action/domain) Declare intent to attend an Event and trigger reminders; updates event capacity. *Performed by: Student | Purpose: Enables event scheduling and attendance management.*

Report / **Block**

(action, domain) Safety-focused controls to restrict visibility or escalate moderation. *Performed by: Student / Recruiter / System | Purpose: Preserves trust, safety, and platform integrity.*

Provide Feedback

(event, domain) Share status, evaluation, guidance, or closure between recruiter and candidate. *Performed by: Recruiter / System | Purpose: Improves transparency and reduces uncertainty.*

Domain Constraints / Rules (invariant business conditions)

Single active offer per application

Prevents contradictory decisions.

Frozen artifacts after Apply

Preserves evidence used in triage.

Mutual interest required for Match

No implicit or unilateral matching.

Visibility always respected

Profiles not shown outside configured scope.

Offer validity window must exist

Does not allow open-ended offers.

Immutable decision history

Notes and interview outcomes cannot be overwritten.

Scope and actors

The domain covers how recruiters discover candidates, evaluate evidence of fit, and make time-bound decisions. It also covers how students prepare and publish profiles and artifacts, apply to openings, communicate with recruiters, and respond to decisions. Primary actors are students, recruiters, and organizations. Secondary actors are career offices and third-party services that send notifications or store artifacts.

Core flow of a hiring cycle

1. A recruiter defines an opening with role, eligibility, skills, seniority, location rules, and a clear decision calendar.
2. Students prepare a profile and publish artifacts such as resume, projects, certifications, and availability.
3. Students submit an application to an opening. The application freezes the versions of the artifacts used for that opening.
4. Recruiters triage the queue of applications using quick signals such as eligibility, program, graduation term, skills match, portfolio completeness, and recent activity.
5. Selected students move to screening and interviews. Interview outcomes and notes accumulate as evidence tied to the same application.
6. Recruiters decide. Outcomes can be rejection, waitlist, or offer. An offer specifies deadline, compensation ranges or bands, start date window, and required actions.
7. Students accept, decline, or ask for more time. The system records a final state for the application and closes the loop with both sides.

Key entities and relationships

Module Organization (Lecture Topic Task: Modules for Profile and Matching)

To improve maintainability, domain clarity, as well as alignment with the principles of Domain Driven Design, the system has been organized into two primary modules: **Profile** and **Matching**. These modules group related classes and behaviors together, each with a clearly defined responsibility within the application domain.

Profile Module

This module encapsulates all functionality related to user identity and profile management. It defines how both students and recruiters represent themselves within the system and provides mechanisms for storing, displaying, and updating profile data.

Included Classes: `StudentProfile` `RecruiterProfile` `Resume` `MediaUpload` `ProfileController`

Core Responsibilities:

- Handle the creation, modification, and persistence of profile data for all users.
- Manage resume and media uploads associated with each profile.
- Ensure that user information is stored and retrieved consistently across the application.
- Provide limited, read only access to profile data through a defined interface, `IProfileReader`, for external modules such as Matching.

The Profile module acts as the source of truth for user related data and serves as an independent subsystem that can evolve without affecting other parts of the domain.

Matching Module

The Matching module defines the behavior that connects students and recruiters through an intelligent matching process. It consumes data from the Profile module to perform its operations but maintains its own internal logic and data flow.

Included Classes: `Match` `MatchService` `MatchingPolicy` `RecommendationEngine`

Core Responsibilities:

- Execute the matching algorithm that pairs students with recruiters based on skill, preference, and availability criteria.
- Manage match creation, updates, and persistence.
- Notify users when a match occurs and handle match related interactions.
- Implement and refine matching policies that determine how compatibility is calculated.

The Matching module does not modify or persist profile data directly. Instead, it retrieves read only information from the Profile module through the `IProfileReader` interface to maintain a clean separation of concerns.

Module Boundaries and Dependency Flow

To maintain loose coupling and modularity, the dependency flow between modules follows a one directional structure: Profile → exposes → IProfileReader, while Matching → uses → IProfileReader.

This ensures that the Matching module depends only on the data abstractions provided by Profile, while Profile remains independent of Matching. As a result, changes in the matching logic do not affect the internal behavior of the Profile module.

Benefits of Modular Design

This modular separation strengthens system scalability and future adaptability. For example:

- The Profile module can evolve to include new profile types or attributes without affecting matching logic.
- The Matching module can introduce new algorithms or policies without impacting profile management.
- Code review and testing processes become simpler due to clearer ownership boundaries.

Through this structure, the system adheres to the principles of high cohesion and low coupling, making sure that each part of the domain remains understandable, testable, and independently maintainable.

Relationship	Multiplicity	Notes
Student to Application	one to many	A student may submit many applications. An application belongs to one student.
Opening to Application	one to many	An opening receives many applications. An application targets one opening.
Application to Interview	one to many	Each interview is tied to one application and records stage, outcome, and notes.
Application to Offer	zero or one	At most one active offer per application. Historical offers remain as records.
Offer to Acceptance	zero or one	One acceptance closes the offer. Decline also closes the offer.
Student to Artifact	one to many	Artifacts are versioned. An application references the versions used at submit time.
Recruiter to Opening	one to many	A recruiter may own several openings across teams or time.
Notification to Event	many to one	Multiple notifications can be sent for a single domain event with different channels.

Invariants that guide design

- An application always links to exactly one student and one opening.
- An offer cannot exist without an application in a decision-eligible state.
- Once an offer is accepted, the application moves to hired and no other offers can be issued for that application.
- Deadlines are stored with time zone and source. Any change to a deadline keeps a trace of who changed it, when, and why.
- Interview outcomes and notes are immutable records once submitted. Corrections are stored as new records that supersede older ones.
- Notifications are reproducible. Given an event and a preference set, the system can explain which messages went out, to whom, and when.

Concrete examples from raw observations

- First triage by recruiters often takes less than one minute and checks basic eligibility and red flags such as missing graduation date or visa requirement.
- Students reuse the same resume across many openings. The application must keep the exact file seen during triage even if the profile later changes.
- Interviewers rely on a daily view named Interview Today with candidate, role, time window, location or link, and a quick link to notes.
- Offers require reminders at common timing windows such as T-24 hours and T-2 hours before the deadline.
- Career offices request an audit record of all messages sent to a student, including channel and delivery status.

Edge cases and ambiguity resolution

- A student accepts after the deadline because a recruiter granted an extension by email. The system records an extension event with the new limit and the actor who granted it.
- A recruiter publishes an offer with a deadline that is too early. The correction updates the active deadline and preserves the original as an error record. All related reminders are recalculated.
- A student submits two applications to the same opening through different channels. Duplicate detection flags the situation and asks the recruiter to merge or keep separate with a reason.
- An opening is withdrawn after interviews due to budget freeze. All active applications move to closed by employer with a reason code and a message to candidates.
- A student withdraws an application after receiving an external offer. The application state becomes withdrawn by candidate and future notifications stop.
- A student updates a resume after applying. The application still shows the submitted version and also displays that a newer profile exists for transparency.
- Recruiter reassignment happens mid-process. Ownership moves to a new recruiter while preserving the decision trail and permissions on notes.

Language and abstractions used consistently

- Ubiquitous terms include Student, Opening, Application, Interview, Offer, Acceptance, Reminder, and Notification. These are treated as core domain nouns reused across 2.1.1, 2.2.1, and 2.2.2.
- Application is the aggregate root for interviews, offer, acceptance, and decision notes. All changes that affect the decision state go through the application.
- Offer Deadline is a value that carries time zone and precision to minutes.
- Artifact Version captures the exact resume or portfolio snapshot attached to an application.
- Triage View and Interview Today are application services that orchestrate domain data into the screens recruiters use.

Why this structure matters?

This narrative ties the abstract model to observable work. The multiplicities clarify what can exist at the same time. The invariants prevent silent corruption such as orphaned offers or moving deadlines without trace. The edge cases show where business rules bend and how the system should keep truth and history. The language aligns with the rough sketch and the terminology so that design, code, and tests refer to the same concepts.

2.1.3. Events, Actions, and Behaviors

The following are concrete examples of events, actions, and behaviors derived from the domain observations. Each example includes a description of the type, triggering conditions, subsequent actions, and postconditions to illustrate how these elements manifest in the student-recruiter interaction context.

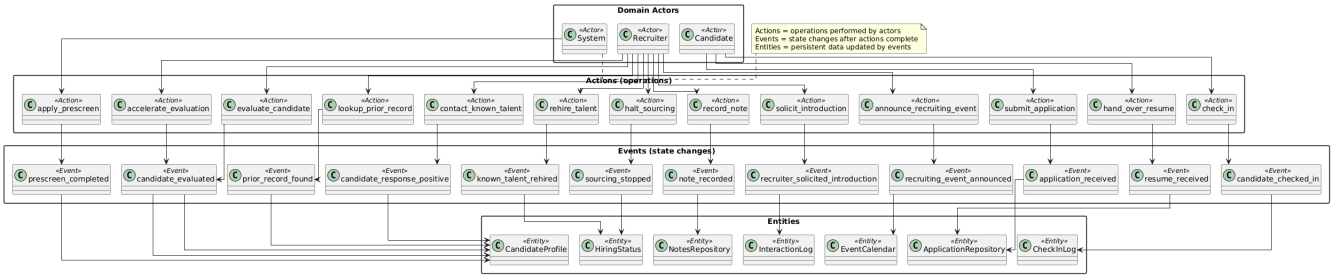


Figure 2.3.1 - 1. UML Diagram of Events, Actions, and Entities

Recruiting event announced (event → actions → follow-up events)

- **Type:** event
- **Event:** A recruiting event has just been announced.
- **Triggered by:** Recruiter, university, or event organizer.
- **Actions following the event:**
  - Recruiter defines roles, basic requirements, and target profiles.
  - Organizer distributes event information through email, flyers, portals.
- **Postcondition:** Candidates become aware of opportunities and plan attendance or early applications.
- **Reference:** Desire to pre-screen before the fair.

Candidate checked in (event interleaved with action)

- **Type:** event
- **Event:** Candidate has just arrived and checked in.
- **Action:** Candidate registers, receives map and agenda.
- **Postcondition:** Candidate is now able to navigate booths and interact with recruiters.

Queue formed in front of booth (event explained in flow)

- **Type:** event
- **Event:** A queue has formed in front of a booth due to multiple candidate arrivals.
- **Actions:** Staff organizes line, candidates wait and skim notes or portfolios.
- **Postcondition:** Long waits → reduced interaction quality.

Recruiter solicited introduction (action → event sequence)

- **Type:** event (solicitation) paired with action (pitch delivery)
- **Event:** Recruiter has just requested an introduction from the candidate.
- **Action:** Candidate delivers a short pitch; recruiter asks fast clarifying questions.
- **Postcondition:** First impression formed under noisy, rushed conditions.

Resume/portfolio handed over (action → event + entity appearance)

- **Type:** hybrid
- **Action:** Candidate hands over resume/portfolio.
- **Event:** Resume/portfolio has just been received by recruiter.
- **Entity:** Resume/portfolio enters a pile, folder, or digital queue.
- **Postcondition:** Document becomes part of a review set and may lose visibility.

Recruiter recording notes (operation → event clearly separated per feedback)

- **Type:** operation + event
- **Operation (action):** Recruiter writes quick notes immediately after interacting with the candidate ("has Python", "not ready").
- **Event:** A note has just been recorded and stored in the notes repository.
- **Postcondition:** Candidate now has tag-linked metadata enabling future recall.
- **Clarification from feedback:** Distinguishes **the act of writing** from **the event of having completed the note**.

Informal meetup occurred (event with embedded actions)

- **Type:** event
- **Event:** An informal or accidental meetup has just occurred outside the booth.
- **Actions:** Short exchange, recruiter asks spontaneous questions, candidate improvises answers.
- **Postcondition:** Lead or opportunity may arise spontaneously.

Multichannel application submitted (action → event → entity)

- **Type:** hybrid
- **Action:** Candidate submits applications across multiple portals.

- **Event:** Application submission has just occurred (possibly in duplicate).
- **Entity:** Application record stored in multiple systems.
- **Postcondition:** Recruiters encounter duplicate or inconsistent entries.

Pre-screen completed (operation → event)

- **Type:** event
- **Operation:** System or recruiter runs heuristics or filters on candidate data.
- **Event:** Pre-screening has just been completed.
- **Postcondition:** Candidate marked “eligible” or “ruled out."

Invitation/follow-up delayed (event contextualized)

- **Type:** event
- **Event:** A follow-up or invitation is now delayed beyond expected timeline.
- **Actions:** Recruiter eventually contacts candidate after backlog clears.
- **Postcondition:** Candidate may have lost interest or accepted another offer.

Decision to stop sourcing (operation → event)

- **Type:** operation + event
- **Operation:** Recruiter analyzes applicant pool and decides to stop sourcing.
- **Event:** Decision to halt sourcing has just been officially made.
- **Postcondition:** Window for new applicants narrows.

Known talent rehired (behavior expanded per professor’s request)

- **Type:** behavior (multi-step pattern composed of operations + events)

Sequence of actions and events (interleaved as required):
1. **Action:** Recruiter identifies a past intern/employee.
2. **Event:** Past performance record becomes relevant to the current opening.
3. **Action:** Recruiter contacts known talent or flags them internally.
4. **Event:** Candidate responds positively and expresses interest.
5. **Action:** Recruiter accelerates evaluation (shortened interview or direct recommendation).
6. **Event:** Hiring decision is finalized earlier than standard pipeline.

- **Postcondition:** Reduced uncertainty; faster onboarding due to strong trust.

Swag distributed (event + entity)

- **Type:** event + entity
- **Event:** Swag has just been handed out.
- **Entity:** Promotional item representing company brand.

- **Postcondition:** Improves brand recall.

Recruiting system / institution (entity with operational capabilities)

- **Type:** system/entity
- **Capabilities:** Announce events, register check-ins, apply filters, deduplicate entries, store applications, maintain logs.
- **Role in events:** Generates and records domain events such as check-ins, pre-screens, or submissions.

Reference providers (entity)

- **Type:** third-party entity
- **Role:** Provide verification, references, and confirmations.

Resume/application repository (entity + operation)

- **Type:** entity + persistence operation
- **Role:** Stores and links application entries; handles deduplication.

Note-taking repository (entity)

- **Type:** entity
- **Role:** Stores recruiter notes tagged per interaction.

Prescreen operation (operation)

- **Type:** operation
- **Role:** Applies heuristics and filters.

Transfer logs (entity)

- **Type:** entity
- **Role:** Record domain events generated from operations (pre-screen, invitations, etc.).

Transfer repository (entity)

- **Type:** entity
- **Role:** Persistent store for transfer-relevant events and artifacts.

Candidate engagement (behavior with interleaved actions + events per feedback)

- **Type:** behavior — multi-step sequence of actions and events

Interleaved sequence:
1. **Event:** Recruiting event announced.
2. **Action:** Candidate prepares résumé and portfolio.
3. **Event:** Candidate checked in.

4. **Action:** Candidate explores booths and waits in queue.
5. **Event:** Recruiter solicits introduction.
6. **Action:** Candidate delivers pitch.
7. **Event:** Resume/portfolio handed over.
8. **Action:** Candidate later submits applications through portals.
9. **Event:** Application submission recorded.

- **Outcome:** Candidate visibility improves, but duplicates and delays may affect results.

Fast screening and prioritization (pattern / behavior)

- **Type:** behavior / operational pattern
- **Definition / justification:** Recruiters manage large candidate volume by applying simple heuristics and rule-based filters. This is a recurring domain pattern composed of repeated prescreen operations and evaluative events.
- **Constituent actions:** – Define heuristics (year, skills, seniority). – Apply prescreen operation to incoming records. – Flag or tag candidates in note-taking repository.
- **Constituent events:** – Pre-screen just completed. – Recruiter recorded notes.
- **Risks / side effects:** Coarse filters may overlook strong or atypical profiles.

Multichannel application handling and consolidation (behavior)

- **Type:** behavior — system + human coordination
- **Definition / justification:** A domain behavior involving repeated candidate submissions across portals and systematic attempts to deduplicate and merge them into a single authoritative record.
- **Constituent actions:** – Candidate submits across portals. – System deduplicates and merges records. – Recruiter/system links records for traceability.
- **Constituent events:** – Application submitted through multiple portals. – Transfer logs updated.
- **Expected outcomes:** Reduced record duplication, though imperfect deduplication may still cause confusion.

Stopping after an early promising cohort (behavior / decision pattern)

- **Type:** behavior / decision pattern
- **Definition / justification:** A sourcing pattern where recruiters evaluate early applicants and then choose to pause or stop sourcing. This qualifies as a behavior because it integrates evaluation actions with a sourcing-halting operation triggered by recent evaluative events.
- **Constituent actions:** – Evaluate initial cohort of candidates (screening, interviews). – Make sourcing decision (operation). – Halt active sourcing operation.
- **Constituent events:** – Pre-screen just completed for initial cohort. – Decision to stop sourcing was just made.
- **Expected outcomes:** Narrowing of applicant window and reduced diversity of incoming profiles.

- **Justification note:** Clarifies that the behavior is composite, not atomic, since it contains both actions and follow-up triggering events.

Communication delay leading to attrition (behavior)

- **Type:** behavior
- **Definition / justification:** A time-sensitive pattern where delays in outreach lead to candidate attrition. This is composed of scheduling actions followed by delayed-contact events.
- **Constituent actions:** – Queue candidates for later contact (operation). – Send invitations or follow-ups after a delay.
- **Constituent events:** – Invitation or follow-up has been delayed. – Candidate withdraws or accepts another offer.
- **Expected outcomes:** Increased drop-off and reduced conversion rates.

Informal networking conversion (behavior)

- **Type:** behavior
- **Definition / justification:** Unplanned or informal interactions often yield deeper conversation and higher-quality leads compared to rushed booth exchanges.
- **Constituent actions:** – Informal conversation initiated by candidate or recruiter. – Focused follow-up scheduled and logged. – Interview scheduled or lead moved into the pipeline.
- **Constituent events:** – Informal meetup occurred.
- **Expected outcomes:** Higher-quality engagement and stronger lead conversion.

Notes as memory (pattern / entity usage)

- **Type:** pattern / entity usage
- **Definition:** Short recruiter notes act as an external memory for many brief encounters. This pattern combines a persistent entity (notes repository) with recurring tagging operations.
- **Constituent actions:** – Write short tags (e.g., “has Python”). – Link notes to candidate records in the repository.
- **Constituent events:** – Recruiter recording notes.
- **Outcome:** Improved recall and more reliable evaluation when revisiting interactions.

Rehiring based on prior trust (behavior)

- **Type:** behavior
- **Definition / justification:** Recruiters often prioritize previous interns or employees due to familiarity and trust. This pattern combines record lookup with accelerated rehire actions.
- **Constituent actions:** – Lookup past intern/employee records in system. – Prioritize outreach and extend offer more rapidly.
- **Constituent events:** – Known talent has been rehired.
- **Expected outcomes:** Faster hiring decisions and reduced uncertainty.

Brand recall through giveaways (behavior / pattern)

- **Type:** behavior / marketing pattern
- **Definition:** Promotional swag distribution leads to higher brand recall, influencing later candidate engagement.
- **Constituent actions:** – Distribute promotional materials or swag. – Candidate later recalls brand and reengages during outreach.
- **Constituent events:** – Swag or promotional material was handed out.
- **Expected outcomes:** Enhanced brand visibility and improved interaction quality during follow-up.

2.1.4. Function Signatures

~~This section describes the key operations that define the system's behavior. Each function signature is presented with its conceptual explanation, separating domain concepts from UI artifacts.~~

1. `getNextProfile`

- **Domain Function**
  - `getNextProfile(user: User) → (ProfileCursor, ProfileCard)`

Purpose: Returns the next card available to a user based on the domain rule: "The discovery flow has sequential steps." The deck is dynamically generated depending on whether the user is a Candidate or a Recruiter, their preferences, and the active job openings. For a Recruiter, this returns a CandidateCard; for a Candidate, a JobCard. Returns `null` if no profiles remain.

Scenario: Recruiter R calls `getNextProfile` with their current cursor → receives the next CandidateCard → cursor updated for next fetch.

2. `processSwipe`

- **Domain Function**
  - `processSwipe(reviewer: Reviewer, card: ProfileCard, interest: Interest) → (Reviewer, InterestRecord)`

Purpose: Records the reviewer's evaluation of a ProfileCard as Interested or NotInterested. Conceptually, a "Like" is equivalent to Interested, and a "Pass" maps to NotInterested. Updates the reviewer aggregate and persists an InterestRecord.

Scenario: Reviewer R swipes Interested on Card C → R's evaluation history updated → InterestRecord created.

3. `determineMatch`

- **Domain Function**
  - `determineMatch(reviewer: Reviewer, target: User, interestRecord: InterestRecord) → Match?`

Purpose: Checks whether reciprocal interest exists; domain rule: "a match occurs when both have expressed interest." This separates domain logic from UI triggers.

Scenario: Reviewer R likes Candidate A → processSwipe generates InterestRecord → `determineMatch` checks if A already liked R → produces Match.

4. `judgeCandidate`

- **Domain Function**
  - `judgeCandidate(recruiter: Recruiter, candidate: Candidate, newStage: CandidateStage) → (Recruiter, Candidate)`

Purpose: Models a recruiter assigning a candidate a new evaluative stage. CandidateStage = {Undecided, NotInterested, Interested, Invited, OfferMade}. Returns updated aggregates for both recruiter and candidate.

5. `dismissCandidate`

- **Domain Function**
  - `dismissCandidate(recruiter: Recruiter, candidate: Candidate, position: JobOpening) → (Recruiter, Candidate)`

Purpose: Represents the domain action of rejecting a candidate for a specific job opening. Semantically clearer than using `judgeCandidate` with NotInterested alone.

6. `establishConnection`

- **Domain Function**
  - `establishConnection(match: Match, recruiter: Recruiter, candidate: Candidate) → (Recruiter, Candidate, Connection)`

Purpose: Creates a persistent connection entity after a successful match. The connection serves as the context for all one-to-one interactions between participants and houses message history and metadata.

7. `sendMessage`

- **Domain Function**
  - `sendMessage(connection: Connection, sender: User, content: MessageContent) → Connection`

Purpose: Adds a new message to a connection's message history. Operates directly on the domain entity and enforces permissions, ensuring proper inclusion in the conversation.

8. `authenticateUser`

- **Domain Function**
  - `authenticateUser(credentials: UserCredentials) → UserSession`

Purpose: Domain-level authentication: validates credentials and produces a UserSession entity

containing identity and role for authorization. Returns an error result for invalid credentials.

9. `viewProfile`

- **Domain Function**
  - `viewProfile(subject: User, viewer: User) → ProfileView`

Purpose: Returns an authorized view of a user's profile based on domain permissions. Encapsulates business rules determining which attributes can be viewed by whom.

Scenario: Recruiter Reviewing a Candidate

1. Fetch Profile: Recruiter calls `getNextProfile("recruiter_123")` → receives CandidateCard for candidate_abc.
2. Express Interest: Recruiter swipes right → calls `processSwipe("recruiter_123", "candidate_abc", SwipeDirection.like)` → triggers domain action `judgeCandidate(recruiter_123, candidate_abc, Interested)`.
3. Check for Match: System calls `determineMatch("recruiter_123", "candidate_abc", interestRecord)` → returns Match if reciprocal interest exists.
4. Establish Connection: Frontend calls `establishConnection(match, recruiter_123, candidate_abc)` → creates a persistent Connection.
5. Communication: Recruiter sends a message using `sendMessage(connection, recruiter_123, "Welcome to our talent pool!")`.

2.2. Requirements

2.2.1. User Stories, Epics, Features

NOTE	This subsection defines the product scope from a user-value perspective. It organizes the solution into Epics that capture high-level goals and Features that realize those goals in the system.

Abbreviations and ID Conventions

Abbrev.	Meaning
US	User Story: a user-centered need framed as intent and value.
E	Epic: a high-level goal that groups related features and stories.
F	Feature: a concrete capability that realizes part of an epic.
ReqRef	Requirement Reference: one or more requirement IDs influenced by a story/feature.
REQ	Requirement: functional or non-functional specification.

Identifier Formats

Type	Format	Components	Example
User Story ID	`US.AREA.NN`	Functional area (e.g., `PROF`, `SRCH`, `MATCH`, `CHAT`, `EVT`, `NOTIF`, `SAFE`) + sequence.	`US.PROF.01`
Epic ID	`E#`	Epic number or “Epic E#: Title”.	`E1`, `Epic E1: Candidate Profile & Portfolio`
Feature ID	`F#.N`	Epic number + feature sequence.	`F1.1`
Requirement ID	`REQ-AREA-TYPE-NN`	`AREA` + `TYPE` (`F` or `NF`) + number.	`REQ-PRF-F-01`

ReqRef Usage

Placement	Syntax
Same line	`US.PROF.01: Publish a complete profile ReqRef: REQ-PRF-F-01, REQ-PRF-NF-01`
Next line	`ReqRef: REQ-PRF-F-01, REQ-PRF-NF-01`

Epic E1: Candidate Profile & Portfolio

Goal: Present credible competence fast.

Problem/value: Candidates need a concise, verifiable profile that allows recruiters to assess fit within seconds.

Features (F1):

- F1.1 Profile editor: The profile editor captures a candidate’s education, skills, experience, and role interests.
- F1.2 Portfolio artifacts: Candidates can upload portfolio items such as PDFs, public links, and videos, and they can reorder those items.
- F1.3 Visibility and privacy controls: Candidates can set their profile visibility to Public, Match-only, or Private and retain full control over exposure.
- F1.4 Profile completeness indicator: The system displays a completeness indicator that shows progress toward a fully publishable profile.

US.PROF.01: Publish a complete profile ReqRef: REQ-PRF-F-01, REQ-PRF-F-02, REQ-PRF-NF-01

"As a candidate, I want to publish my education, skills, and experience so recruiters can evaluate fit quickly."

This story generates **multiple requirements**: - functional profile publishing logic - validation of required fields - performance expectation for publishing and updating

Acceptance criteria:

Given	When	Then
A verified account	All required fields are completed	The profile is published as “Complete”
Required info missing	Publish attempted	Inline errors indicate missing fields
Profile is updated	Changes saved	Last-updated timestamp refreshed

US.PROF.02: Add portfolio items ReqRef: REQ-PRF-F-03, REQ-PRF-NF-02

"As a candidate, I want to add portfolio items so that my work is easy to review."

This story results in multiple requirements: - upload + type/size validation - performance and reliability constraints

Acceptance criteria:

Given	When	Then
Valid file or URL	Upload occurs	Item appears and can be reordered
Unsupported type/size	Upload attempted	Error lists allowed types/sizes

US.PROF.03: Control profile visibility ReqRef: REQ-PRF-F-04, REQ-PRF-NF-03

"As a candidate, I want to choose my profile visibility so I control my exposure."

This story influences: - access-control requirements - search-filtering behavior - privacy/performance guarantees

Acceptance criteria:

Given	When	Then
Visibility = Private	Recruiters search	Profile is hidden
Visibility = Match-only	A mutual Match occurs	Profile becomes visible
Visibility = Public	Recruiters search	Profile may appear in results

E1 Traceability (Stories → Features → Requirements)

Story	Features	ReqRef	Notes
US.PROF.01	F1.1, F1.4	REQ-PRF-F-01, REQ-PRF-F-02, REQ-PRF-NF-01	One story drives several functional + NF requirements.

Story	Features	ReqRef	Notes
US.PROF.02	F1.2	REQ-PRF-F-03, REQ-PRF-NF-02	Upload constraints + performance.
US.PROF.03	F1.3	REQ-PRF-F-04, REQ-PRF-NF-03	Access-control + privacy logic.

Epic E2: Recruiter Discovery & Search

Goal: Shortlist qualified candidates efficiently.

Problem/value: Recruiters need to discover relevant candidates quickly and understand organizational context without friction.

Features (F2):

- F2.1 Company page: The organization can publish a company page with logo, sectors, locations, and available roles.
- F2.2 Candidate search with filters: Recruiters can search using filters such as skills, role interests, and availability.
- F2.3 Candidate detail view: Recruiters can open a detailed candidate view that consolidates profile and portfolio information.

US.SRCH.01: Publish a company page ReqRef: REQ-SRCH-F-01, REQ-SRCH-NF-01

"As a recruiter, I want to publish a company page so candidates understand who we are and our roles."

One story → multiple requirements: - functional publishing behavior - performance/loading guarantees

Acceptance criteria:

Given	When	Then
Logo/description/sectors provided	Publishing requested	Page becomes visible
Fields incomplete	Publishing requested	Prompts indicate missing fields

US.SRCH.02: Filter and rank candidates ReqRef: REQ-SRCH-F-02, REQ-SRCH-F-03, REQ-SRCH-NF-02

"As a recruiter, I want to filter and rank candidates so I can efficiently identify high-fit profiles."

This story influences: - multiple functional filtering + ranking requirements - non-functional search performance requirements

Acceptance criteria:

Given	When	Then
Filter criteria selected	Search executed	Results reflect filters
Large candidate set	Ranking requested	Sorted results returned within acceptable delay
Invalid filter combination	Search executed	System warns or adjusts filters

E2 Traceability (Stories → Features → Requirements):

Story ID	Feature	ReqRef	Notes
US-SRCH-01	F2.1	REQ-SRCH-F-01	Company profile schema.
US-SRCH-02	F2.2, F2.3	REQ-SRCH-F-02	Filter set and performance target.

Epic E3: Matching & Messaging

Goal: Move from interest to conversation quickly.

Problem/value: Both parties need a fast way to express interest, form a mutual Match, and start secure conversations.

Features (F3):

- F3.1 Like and Pass interactions: Users can register quick likes or passes on presented profiles.
- F3.2 Mutual Match and notification: The system detects mutual interest and triggers an in-app notification that opens a chat.
- F3.3 One-to-one chat: Matched users can exchange messages with delivery and read states.

US.MATCH.01: Express quick interest | ReqRef: REQ-MATCH-F-01

"As a candidate, I want to like or pass quickly so that I can move fast through options."

Acceptance criteria:

Given	When	Then
A profile is shown.	Like is pressed.	Interest is stored.
A profile is shown.	Pass is pressed.	That profile is removed from the current session.

US.MATCH.02: Get notified on mutual like | ReqRef: REQ-MATCH-F-02

"As a user, I want to be notified when there is a mutual like so that a conversation can start."

Acceptance criteria:

Given	When	Then
Both sides liked each other.	The system detects mutual like.	A chat thread opens and an in-app notification is sent.

US.CHAT.01: Exchange messages with safety | ReqRef: REQ-CHAT-F-01

"As a matched user, I want to exchange messages so that next steps can be coordinated."

Acceptance criteria:

Given	When	Then
A Match chat is open.	A message is sent.	The recipient receives it near real time; the sender sees sent and read states.
The other party is blocked.	They attempt to send a message.	The message is not delivered and no notification is generated.

E3 Traceability (Stories → Features → Requirements):

Story ID	Feature	ReqRef	Notes
US-MATCH-01	F3.1	REQ-MATCH-F-01	Interaction logging.
US-MATCH-02	F3.2	REQ-MATCH-F-02	Match detection and notification trigger.
US-CHAT-01	F3.3	REQ-CHAT-F-01	Realtime delivery, receipts, block rules.

Epic E4: Events & Notifications

Goal: Increase attendance and timely follow-through.

Problem/value: Candidates must discover opportunities in time and receive reminders that respect preferences and quiet hours.

Features (F4):

- F4.1 Events feed: The system lists events with title, date and time, location, and RSVP state.
- F4.2 RSVP and reminders: Users can RSVP and receive reminders before the event.
- F4.3 Notification preferences: Users can configure quiet hours and choose preferred channels.

US.EVT.01: Discover and RSVP to events | ReqRef: REQ-EVT-F-01

"As a candidate, I want to see upcoming recruiting events and RSVP so that opportunities are not

missed."

Acceptance criteria:

Given	When	Then
Events feed is available.	-	Items are ordered by date and show title, location, and RSVP.
An RSVP exists.	The event is 24 hours away.	An in-app reminder is delivered.

US.NOTIF.01: Respect notification preferences | ReqRef: REQ-NOTIF-F-01

"As a user, I want notifications to follow my channel and quiet-hour settings so that interruptions are minimized."

Acceptance criteria:

Given	When	Then
Quiet hours are active.	A non-urgent event occurs.	Notifications are queued until quiet hours end.
The user opted in to in-app only.	A reminder must be sent.	Only in-app is used; no email or SMS is sent.

E4 Traceability (Stories → Features → Requirements):

Story ID	Feature	ReqRef	Notes
US-EVT-01	F4.1, F4.2	REQ-EVT-F-01	RSVP state and reminders.
US-NOTIF-01	F4.3	REQ-NOTIF-F-01	Quiet hours and channel policy.

Epic E5: Safety & Moderation

Goal: Maintain a safe, trusted environment.

Problem/value: Users must be able to report issues and block unwanted contacts, and moderators need clear workflows.

Features (F5):

- F5.1 Report a profile: Users can submit a report for moderation review.
- F5.2 Block or unblock a user: Users can block or later restore interaction with another profile.
- F5.3 Moderation review queue: Administrators can triage and process reported cases.

US.SAFE.01: Report inappropriate behavior | ReqRef: REQ-SAFE-F-01

"As a user, I want to report a profile so that moderation can review and act."

Acceptance criteria:

Given	When	Then
A report is submitted.	-	A moderation case is created with timestamp and reporter ID.

US.SAFE.02: Block interactions | ReqRef: REQ-SAFE-F-02

"As a user, I want to block a profile so that it no longer appears or can initiate chats."

Acceptance criteria:

Given	When	Then
Block action is confirmed.	-	The blocked profile no longer appears and new chats cannot be opened.
Unblock is requested.	-	Visibility and messaging return to the pre-block state.

E5 Traceability (Stories → Features → Requirements):

Story ID	Feature	ReqRef	Notes
US-SAFE-01	F5.1, F5.3	REQ-SAFE-F-01	Moderator workflow.
US-SAFE-02	F5.2	REQ-SAFE-F-02	Block and unblock policy with propagation.

2.2.2. Personas

~~**Narrative personas for Milestone 1 & 2:**~~

The personas below represent our core user segments and ground the scope of this product. For each persona we outline goals, pains, typical behaviors, and accessibility needs, and we link them to the stories, epics, and features defined in 2.2.1. We'll reference these personas by name during planning and reviews to keep decisions concrete and tied to user value.

Table 1. María "New Grad" Rivera — University Candidate (mobile-first)

Snapshot	Loves hackathon weekends and cafés near campus; anxious about first-job search but optimistic.
Background	22, UPRM (CS). First-gen grad, part-time tutoring; lives off-campus with roommates.

Motivations	Land her first SWE role where she can keep learning; wants fast, clear signals of interest.
Hobbies	Campus hackathons, short video reels of projects, weekend hikes.
Tech Setup	iPhone as primary device; edits portfolio on a shared laptop.
Relationship to App	Wants quick Like/Pass and reminders for events tied to companies she follows.
Goals	Publish a complete profile quickly; showcase a simple portfolio; control visibility; get event reminders.
Pains	Long forms; vague errors; unwanted exposure.
Behavior	Short sessions; prefers simple actions (Like/Pass).
Accessibility	Clear, actionable error messages; low latency on mobile.
Related Stories	Create profile; Add portfolio; Choose visibility; Like/Pass; Match notification; 1:1 messaging; Events feed & RSVP.
Epics	Candidate Profile & Portfolio; Matching & Messaging; Events & Notifications; Safety & Moderation.
Quote	"I want to upload the essentials and start exploring without oversharing."

Table 2. Luis “Switcher” Santiago — Career-transition Candidate (privacy-first)

Snapshot	Careful planner changing lanes into QA; values control and signal quality over volume.
Background	30, IT support → moving into QA; evening bootcamp; helping family on weekends.
Motivations	Show real, verifiable skills without broadcasting a job search to current contacts.
Hobbies	Keyboard mods, bug-bash meetups, journaling progress.
Tech Setup	Desktop first; tracks opportunities in spreadsheets.
Relationship to App	Prefers “visibility by Match”; wants strong filters and concise profile previews.
Goals	Import/organize history; highlight skills; appear in relevant searches without going fully public.
Pains	Lack of control over who sees his profile; imprecise recruiter filters.
Behavior	Logs in a few times per week; replies only when there’s a real Match.
Accessibility	Desktop-oriented; concise summaries.
Related Stories	Choose visibility (private/by-Match/public); Profile & portfolio; 1:1 messaging.
Epics	Candidate Profile & Portfolio; Matching & Messaging; Safety & Moderation.
Quote	"I want to be visible only to people who truly match with me."

Table 3. Karla “Campus Recruiter” Gómez — Recruiter (events & funnel)

Snapshot	Organized, metric-driven; splits time between campus events and fast triage.
Background	Tech company recruiter; owns 3 junior openings; coordinates campus tours with a small team.
Motivations	Build a clean funnel quickly; reduce no-shows; capture reliable signals pre-event.
Hobbies	Morning runs; mentors student groups; podcast commutes.
Tech Setup	Laptop + ATS tabs; lives in filters and saved searches.
Relationship to App	Needs crisp company page, combined filters, and event RSVP with reminders.
Goals	Publish company page; filter by skills/interest/availability; view candidate detail; manage RSVPs and reminders.
Pains	Noisy results; search latency; incomplete candidate info.
Behavior	1–2 h desktop sessions; heavy use of combined filters; saves shortlists.
Service Level	Search with sample data should load in ~2s (p95).
Related Stories	Company page; Search with filters; Results highlight matched terms; Events feed & RSVP; Notifications.
Epics	Recruiter Discovery & Search; Events & Notifications.
Quote	"I need ten viable profiles in minutes and a way to nurture them to interview."

Table 4. Jorge “HR Generalist” Ortiz — SMB Recruiter (speed & safety)

Snapshot	Wears many hats; wants quick, safe conversations that don’t waste cycles.
Background	HR at a 35-person firm; manages onboarding, payroll, and recruiting.
Motivations	Shortlists fast; protect team time; keep the conversation professional and safe.
Hobbies	Weekend leagues; DIY home projects.
Tech Setup	Older office desktop; checks mobile during site visits.
Relationship to App	Needs practical filters, read receipts, and easy report/block.
Goals	Filter by skills and availability; chat 1:1; report or block bad behavior.
Pains	Incomplete profiles; spam/inappropriate contacts.
Behavior	Short work blocks; values online indicators and read receipts.
Related Stories	Filter by skills/interest/availability; 1:1 chat with sent/read states; Report/Block.
Epics	Recruiter Discovery & Search; Matching & Messaging; Safety & Moderation.
Quote	"Give me a reliable shortlist and a clear conversation; the rest is noise."

Table 5. Ana “Safe User” Lozada - Safety-focused Candidate (safety-first)

Snapshot	Cautious first-timer; wants control and predictable notifications.
Background	24, first time on a jobs platform; previous negative social app experiences.
Motivations	Try a new channel without risking privacy or overwhelm.
Hobbies	Photography walks, language exchange groups.
Tech Setup	Android mid-range; limits notifications outside 9–6.
Relationship to App	Wants visibility controls, block/report, and meaningful alerts only.
Goals	Block or report profiles; prevent re-appearance after Pass; receive only useful notifications.
Pains	Unwanted interactions; intrusive alerts.
Behavior	Reviews privacy settings; uses reporting if something feels unsafe.
Accessibility	Simple controls for privacy, block, and report.
Related Stories	Report/Block; Like/Pass does not re-show in session; Relevant in-app notifications.
Epics	Safety & Moderation; Matching & Messaging; Events & Notifications.
Quote	"I want to feel in control and safe at all times."

Table 6. Mina “International Grad” Shah - International Candidate (compliance-first)

Snapshot	International MS grad navigating visas and time zones; needs clarity and eligibility signals.
Profile	24, MS in Data Science, international student; lives off-campus; phone for browsing, laptop for uploads.
Goals	Visa-friendly profile & portfolio; appear in searches filtered by skills, location, and authorization; timely Match notifications; RSVP and reminders; safe messaging.
Pains	Ambiguous job location and start date; unclear offer validity; outreach without consent; slow search; duplicate event notices.
Behavior	Curates projects weekly; short-burst swipes; evening chats; shortlists companies; RSVPs to virtual events.
Accessibility	Clear copy on compensation (salary + benefits), readable tables, timezone-aware reminders.
Related Stories	Create profile & portfolio; Choose visibility; Recruiter search (skills, location, authorization, availability); Match notification; 1:1 messaging; Events feed & RSVP; Report/Block.
Epics	Candidate Profile & Portfolio; Recruiter Discovery & Search; Matching & Messaging; Events & Notifications; Safety & Moderation.
Quote	"Make it crystal clear where the role is, whether I’m eligible, and ping me when it’s a real match—then I can move fast."

Coverage matrix (personas × epics)

Persona	Candidate Profile & Portfolio	Recruiter Discovery & Search	Matching & Messaging	Events & Notifications	Safety & Moderation
María (New Grad)	X		X	X	X
Luis (Switcher)	X		X		X
Karla (Recruiter)		X		X	
Jorge (HR Gen.)		X	X		X
Ana (Safe User)			X	X	X
Mina (Intl. Grad)	X	X	X	X	X

Narrative Personas

The following personas reinterpret our earlier feature-focused profiles as vivid, system-independent characters drawn from the domain. Each one should feel like a real person whose name can act as shorthand for their wishes, goals, fears, and decision habits

Table 7. María "New Grad" Rivera - Narrative Persona

Snapshot	María is the friend who rushes across campus with a coffee in one hand and a folder of résumés in the other, squeezing job-hunt moments into the ten minutes between class, tutoring, and the last shuttle home.
Background	22-year-old CS student at UPRM, first in her family to graduate. She juggles a capstone project, a part-time tutoring job, and calls from home about scholarships and family bills.
Wishes & Goals	Wants her first software role to be a place where she keeps learning and is taken seriously, not "just another intern". Hopes to earn enough to help her parents breathe easier while still having time for side projects and friends.
Fears & Pains	Still remembers waiting nearly forty minutes in a noisy job-fair line only to hand over a résumé and never hear back. She fears becoming invisible in big applicant pools and worries that one awkward conversation might erase months of preparation.
Decision Tendencies	Makes short handwritten lists of companies she truly cares about and circles those where recruiters seemed genuinely interested. She says yes quickly when communication is specific and respectful, but silently drops opportunities that ghost her or feel transactional.

Recruiting Habits	Uses campus career fairs, WhatsApp chats with classmates, and LinkedIn posts to spot openings. After each interaction she writes down the recruiter’s name, what they mentioned, and any promised follow-up so she does not lose track.
Tech & Environment	Checks roles on her phone while walking between buildings or waiting for the bus; deeper work such as tailoring applications happens late at night on a shared, occasionally unreliable laptop at the kitchen table.
Visual Cue	María in worn sneakers at the student center, backpack half-open, a highlighter in hand tracing circles around three companies on a folded job-fair map.
Domain Role	Embodies the Student / Applicant who needs visibility, timely feedback, and fair early-career chances without having to master every tool or platform at once.

Table 8. Luis "Switcher" Santiago - Narrative Persona

Snapshot	Luis is the meticulous coworker who always knows which server is failing and which cable belongs where, quietly studying for a new career after everyone else has logged off.
Background	30-year-old support technician who has spent years fixing other people’s outages. He is halfway through an evening QA bootcamp and often helps his younger cousins with homework on weekends.
Wishes & Goals	Wants to prove that his hands-on troubleshooting experience counts as much as formal titles. His goal is a stable QA role with room to grow, where he can stop hiding his job search from current colleagues.
Fears & Pains	Fears that recruiters will see only "IT support" and never the disciplined tester underneath. He is anxious about his manager discovering that he is interviewing elsewhere and about being forced to accept the first mediocre offer out of fear.
Decision Tendencies	Researches obsessively before applying, saving notes in color-coded spreadsheets. He responds only to opportunities that explain expectations, compensation range, and work modality clearly; vague descriptions go into a "maybe someday" tab that he rarely reopens.
Recruiting Habits	Prefers asynchronous communication — email, messages he can answer late at night, or quiet one-on-one calls after work. He avoids anything that feels like a public announcement of his search and looks for signals that companies respect privacy and confidentiality.
Tech & Environment	Works on a home desktop with multiple monitors, keeping browser tabs for bootcamp materials, test reports, and job leads side by side. On the commute he reviews saved roles on his phone but rarely applies from it.
Visual Cue	Luis at his desk after hours, office lights mostly off, bootcamp slides on one monitor and a spreadsheet of "target roles" open on the other.

Domain Role	Represents the Candidate already in the workforce who values control over visibility, accurate filters, and verifiable signals more than sheer volume of outreach.

Table 9. Karla "Campus Recruiter" Gómez - Narrative Persona

Snapshot	Karla is the recruiter who arrives at campus with a rolling suitcase of banners, a thermos of coffee, and a mental checklist of ten hires she needs before the end of the quarter.
Background	Mid-career recruiter at a tech company responsible for several junior roles. She coordinates campus visits, tracks dozens of conversations in parallel, and reports weekly numbers to her hiring managers.
Wishes & Goals	Wants to leave each event with a short, trustworthy list of candidates she can confidently present to engineering leads. She values reliability over spectacle and hates wasting time on incomplete profiles.
Fears & Pains	Fears promising candidates slipping through the cracks because their names get lost in spreadsheets or handwritten notes. She worries about no-shows at interviews and about overpromising timelines she cannot keep.
Decision Tendencies	Thinks in funnels and stages: discover, screen, interview, offer. She prioritizes candidates who show up prepared, follow through after conversations, and communicate clearly about availability.
Recruiting Habits	Combines university rosters, event RSVPs, and her own notes to plan outreach. During a fair she scans badges quickly, jotting shorthand tags ("data viz", "strong communicator") and marking a star next to those she wants to contact within 48 hours.
Tech & Environment	Lives in tabs: ATS, spreadsheets, shared calendars, and video-conference links. She often works on a laptop balanced on a small booth table, tethered to spotty campus Wi-Fi.
Visual Cue	Karla standing behind a company banner, a lanyard full of badges and a stack of color-coded sticky notes on the table beside her laptop.
Domain Role	Embodies the Recruiter who needs efficient triage, reliable evidence of skills, and tools that preserve memory across many brief student encounters.

Table 10. Jorge "HR Generalist" Ortiz - Narrative Persona

Snapshot	Jorge is the one-person HR department at a small firm, answering benefits questions, fixing payroll issues, and interviewing candidates all in the same afternoon.
Background	Works at a 35-person company where he handles onboarding, policy questions, and recruiting for interns and entry-level roles. Formal tools are limited; much of the process lives in his head and a few shared documents.

Wishes & Goals	Wants a shortlist of dependable candidates who will stay and grow with the team, not just fill seats. He values professionalism, clear communication, and low drama over fancy résumés.
Fears & Pains	Fears hiring someone who leaves after a few months because expectations were misaligned. He dreads spammy messages and candidates who treat interviews casually, because every wasted slot eats into time needed for payroll and compliance work.
Decision Tendencies	Schedules interviews in tight blocks, preferring straightforward conversations. If a candidate communicates respectfully, arrives prepared, and responds quickly to follow-ups, Jorge is inclined to move fast with an offer.
Recruiting Habits	Relies on referrals, local universities, and a small number of online postings. Keeps notes about each conversation in a simple document and uses flags in his inbox to remember who needs a response.
Tech & Environment	Works on an older desktop during office hours and checks email on his phone while walking between meetings or visiting different sites.
Visual Cue	Jorge at a cluttered desk with a stack of onboarding folders on one side and a printed list of interview times clipped to a clipboard on the other.
Domain Role	Represents the Employer-side Recruiter / HR Generalist who values speed, safety, and clear signals more than elaborate workflows.

Table 11. Ana "Safe User" Lozada - Narrative Persona

Snapshot	Ana is the cautious friend who double-checks every privacy setting before joining a new group chat and who would rather miss an opportunity than feel unsafe.
Background	24-year-old recent graduate entering the job market after a few unsettling experiences on social media platforms. She lives with roommates, shares Wi-Fi, and often studies in public spaces like libraries and cafés.
Wishes & Goals	Wants to explore professional opportunities without being overwhelmed by unsolicited messages or pressured into sharing more personal information than necessary.
Fears & Pains	Worries about harassment, fake offers, and her personal data being reused without consent. Even legitimate notifications can feel invasive if they arrive late at night or interrupt time with family.
Decision Tendencies	Reads policies and settings carefully, turning off most channels by default. She stays engaged with platforms that respect her boundaries and make it easy to report or block inappropriate behavior.
Recruiting Habits	Prefers written communication that leaves a trace and lets her think before replying. She appreciates recruiters who introduce themselves clearly, explain why they reached out, and give her options instead of ultimatums.

Tech & Environment	Uses a mid-range Android phone as her main device, often on battery-saver mode, and checks messages in set windows during the day rather than continuously.
Visual Cue	Ana on a quiet balcony in the evening, phone in hand, carefully toggling notification switches before tapping "save".#
Domain Role	#Embodies the Safety-focused Candidate whose trust depends on predictable notifications, clear controls, and effective moderation paths.

Table 12. Mina "International Grad" Shah - Narrative Persona

Snapshot	Mina is the international student who always carries a folder of immigration documents in her backpack, just in case, and tracks time zones across three countries on her watch.
Background	24-year-old MS student in Data Science studying abroad. She balances thesis work, part-time research assistance, and late-night video calls with family in another time zone.
Wishes & Goals	Wants a role where her visa status is understood and respected, and where she can apply her analytics skills to real problems rather than short-term gigs. She hopes to plan her life several months ahead instead of living deadline to deadline.
Fears & Pains	Lives with constant low-level anxiety about whether a role is actually visa-eligible, what the real compensation package looks like, and how long she has to respond to an offer before it expires.
Decision Tendencies	Creates comparison charts of openings including location, work modality, sponsorship options, and start-date flexibility. She commits only after confirming that expectations match on both sides.
Recruiting Habits	Attends virtual career fairs, follows specific employers that explicitly mention sponsorship, and keeps screenshots of any written confirmation about eligibility or deadlines.
Tech & Environment	Uses her phone for quick scans of opportunities during commutes and a laptop in the evenings to update portfolios, prepare case studies, and schedule interviews across time zones.
Visual Cue	Mina at a shared kitchen table late at night, laptop open to a calendar full of color-coded time-zone labels and a stack of printed offer letters with sticky notes marking key clauses.
Domain Role	Represents the International Candidate navigating authorization rules, time zones, and high-stakes decisions that depend on precise information about location, compensation, and validity periods.

2.2.3. Domain Requirements

The domain requirements are the fundamental rules and constraints of the problem space, independent of the software implementation.

ID	Description	Linked User Stories/Epics	Verification Tests
DR001	The system must uniquely identify every candidate, recruiter, employer, and event organizer, distinguishing natural persons from institutional profiles.	US-Auth-001: Register as Candidate, US-Auth-002: Register as Recruiter	T-DB-001, T-API-001
DR002	The system must record the role performed by each entity in every interaction, along with the time and context of that role.	US-Interaction-001: Express Interest, US-Interaction-002: Create Match	T-Log-001, T-DB-002
DR003	The system must represent every opening with explicit requirements (min. experience, qualifications, certifications, languages, legal authorizations, work modality, location, compensation range, and start date).	Epic-JobMgmt	T-API-002, T-UI-001
DR004	The system must record the number of available positions for each opening.	US-Recruiter-001: Post New Opening	T-DB-003
DR005	The system must link every opening to its responsible employer entity.	US-Recruiter-001: Post New Opening	T-API-003
DR006	The system must allow requirement updates for an opening.	US-Recruiter-002: Update Opening	T-Audit-001
DR007	The system must preserve a complete history of all requirement changes for an opening (audit trail).	US-Recruiter-002: Update Opening	T-Audit-001
DR008	The system must represent each candidate profile with education, work history, competencies, certifications, portfolios, and verifiable references.	Epic-ProfileMgmt	T-DB-004
DR009	The system must allow documents to be attached to the candidate profile.	US-Candidate-001: Upload Resume	T-File-001
DR010	The system must preserve the issue date, validity, and verification status for all attached documents.	US-Candidate-001: Upload Resume	T-File-001
DR0011	The system must record candidate preferences such as areas of interest, desired location, and work modality.	US-Candidate-002: Set Preferences	T-DB-005
DR012	The system must represent recruiting events with name, organizer, venue, agenda, date, and capacity.	Epic-EventMgmt	T-DB-006, T-API-004

ID	Description	Linked User Stories/Epics	Verification Tests
DR013	The system must represent company booths/tables within an event structure.	Epic-EventMgmt	T-DB-006, T-API-004
DR014	The system must register the attendance and arrival of candidates and recruiters at an event.	US-Event-001: Check-in at Event	T-Event-001
DR015	The system must enforce event registration rules and capacity limits upon check-in.	US-Event-001: Check-in at Event	T-API-005
DR016	The system must model queues in front of event booths/tables.	Epic-QueueMgmt	T-Queue-001
DR017	The system must enforce a first-come, first-served order for modeled queues.	Epic-QueueMgmt	T-Queue-001
DR018	The system must allow limiting the duration of each queue turn.	US-Event-002: Manage Queue	T-Queue-002
DR019	The system must register the closure of each turn.	US-Event-002: Manage Queue	T-Queue-002
DR020	The system must prevent assigning turns that would exceed declared capacity or time limits.	US-Event-002: Manage Queue	T-Queue-002
DR021	The system must allow candidates and recruiters to express positive or negative interest in openings or profiles.	US-Interaction-001: Express Interest	T-API-006
DR022	The system must record reciprocal expressions of interest.	US-Interaction-001: Express Interest	T-API-006
DR023	The system must create a Match record when both sides express positive interest.	US-Interaction-002: Create Match	T-Match-001
DR024	The system must preserve the date and context of every Match creation event.	US-Interaction-002: Create Match	T-Match-001
DR025	The system must allow recruiters to define shortlisting criteria and register the decision.	US-Recruiter-003: Shortlist Candidate	T-API-007
DR026	The system must record the justification for every shortlisting or rejection decision.	US-Recruiter-003: Shortlist Candidate	T-API-007
DR027	The system must allow candidates to withdraw their interest from an opening or Match.	US-Candidate-003: Withdraw Interest	T-Match-002
DR028	The system must automatically update any pending Matches when interest is withdrawn.	US-Candidate-003: Withdraw Interest	T-Match-002

ID	Description	Linked User Stories/Epics	Verification Tests
DR029	The system must represent the availability of candidates and recruiters through calendars and time blocks.	Epic-Scheduling	T-DB-007
DR030	The system must schedule interviews only for valid, active Matches.	US-Scheduling-001: Book Interview	T-Schedule-001
DR031	The system must prevent double booking of time slots for interviews.	US-Scheduling-001: Book Interview	T-Schedule-001
DR032	The system must register interview outcomes with clear states (continues, rejected, offer extended) and record the date and responsible party.	US-Scheduling-001: Book Interview	T-Schedule-001
DR033	The system must allow message exchanges between candidate and recruiter.	US-Communication-001: Send Message	T-Comm-001
DR034	The system must restrict message exchanges to only valid Matches or where explicit candidate permission is granted.	US-Communication-001: Send Message	T-Comm-001
DR035	The system must register candidate consent for sharing information with an employer or organizer.	US-Candidate-004: Manage Consent	T-DB-008
DR036	The system must allow the revocation of candidate consent for information sharing.	US-Candidate-004: Manage Consent	T-DB-008
DR037	The system must compute the degree of requirement fulfillment for each candidate against an opening (meets, partially meets, does not meet).	US-Recruiter-004: View Match Score	T-Algo-001
DR038	The system must display the computed degree of requirement fulfillment to the recruiter.	US-Recruiter-004: View Match Score	T-Algo-001
DR039	The system must generate a notification when a Match is created.	US-Notification-001	T-Notif-001
DR040	The system must generate a notification when an interview is confirmed.	US-Notification-001	T-Notif-001
DR041	The system must generate a notification when changes to opening requirements affect a candidate’s eligibility.	US-Notification-001	T-Notif-001
DR042	The system must notify candidates about recruiting events related to employers or openings in which they have shown interest.	US-Notification-002	T-Notif-002

ID	Description	Linked User Stories/Epics	Verification Tests
DR043	The system must record domain metrics such as match rate, queue waiting time, average turn duration, follow-up rate, and event attendance.	Epic-Reporting	T-Metric-001
DR044	The system must support funnel analysis from expression of interest to offer extended.	Epic-Reporting	T-Metric-001

2.2.4. Interface Requirements

Interface requirements define how the system interacts not only with users but also with external systems, environmental phenomena, and any data or events crossing the system boundary. This includes user interfaces, system-to-system interfaces, timing constraints, and backend-visible interactions.

ID	Description	Linked User Stories/Epics	Verification Tests
IR001	The system must provide a structured registration interface for new recruiters, including fields for company name, professional email, and job role.	US-Auth-002: Register as Recruiter	T-UI-010, T-FE-001
IR002	The initialization process must include a validation step where a confirmation email is sent, and the internal profile must not be active until validation is complete.		T-BE-001, T-Email-001
IR003	The system must provide an edit profile screen for students to update their skills/competencies (e.g., adding new projects).	US-Candidate-005: Update Skills	T-UI-011
IR004	The internal skill representation must be updated within a maximum of 300 ms after the user saves changes, ensuring a measurable, achievable update time instead of an absolute immediate requirement.		T-BE-002, T-Perf-001
IR005	The system must present a visual notification to both users within 500 ms of a Match being recorded.	US-Notification-001	T-UI-012, T-Notif-003
IR006	The Match notification interface must clearly indicate who the Match is with and include a prominent means to initiate communication.		T-UI-013

ID	Description	Linked User Stories/Epics	Verification Tests
IR007	The system must provide a company profile management screen for authorized recruiters to correct company information (e.g., description, website URL).	US-Recruiter-005: Update Company Info	T-UI-014
IR008	The system must log all changes made via the company profile management screen.		T-Log-002
IR009	The company name field shall be immutable through the company profile management screen after initial validation; changes must be handled by a separate administrative process.		T-FE-002, T-Security-003
IR010	The system must update the internal state within 200–350 ms to reflect a user's swipe action on a profile card (recording the decision and removing the card), ensuring the round-trip time for card replacement meets performance expectations.	US-Interaction-001: Express Interest	T-FE-003, T-Perf-002
IR011	The system must expose a backend interface to retrieve updated profile-card batches, supporting pagination, caching hints, and request timestamps.	US-Interaction-002	T-API-001
IR012	The system must interpret environmental events relevant to the platform - for example, backend-detected account flagging, admin interventions, or automatic expiration of stale matches - and reflect these changes in the user interface within 1 second.		T-BE-004, T-UI-015
IR013	The system must provide an interface for the recommendation algorithm to push updated candidate rankings into the frontend, with a maximum push-latency of 400 ms.	EP-Recommendation-001	T-Rec-001
IR014	The system must provide integration endpoints for institutional authentication (e.g., campus login providers), including OAuth2-based token exchange and error-handling interfaces.	US-Auth-003	T-Security-004, T-API-002

ID	Description	Linked User Stories/Epics	Verification Tests
IR015	The system must expose a public timestamped event stream (internal-only) that allows admin dashboards to reflect live metrics such as active matches, swipe volume, and onboarding progress.	EP-Admin-001	T-Event-001
IR016	The system must provide visual and programmatic error interfaces for network failures, fallback states, or unavailable profile data, ensuring consistent communication of partial-service conditions.		T-UI-016, T-API-003
IR017	The system must reflect environmental constraints such as rate limits, access control decisions, and backend throttling by communicating appropriate messages or delays to the user within 250 ms.		T-Perf-003, T-Security-005
IR018	When external job sources (e.g., company ATS integrations or internship feeds) update recruiter data, the system must reflect these changes in the recruiter's interface within 2 seconds.	EP-ExternalSync-001	T-Sync-001

2.2.5. Machine Requirements

Machine requirements specify the hardware, software, and environment needed for development, testing, and deployment.

ID	Description	Linked User Stories/Epics	Verification Tests
MR001	Developer workstations must meet a minimum specification of Dual-core processor (Intel i5/AMD equivalent), 8 GB RAM, 256 GB SSD, and GPU capable of supporting Flutter/Android/iOS emulators efficiently.	Epic-DevSetup	T-Env-001
MR002	Mobile devices for testing must include Android devices running Android 10.0+ with at least 3 GB RAM and iOS devices running iOS 15+ (e.g., iPhone 11 or newer). Devices must support automated UI testing frameworks where applicable.	Epic-Testing	T-Env-002

ID	Description	Linked User Stories/Epics	Verification Tests
MR003	Servers / Cloud Hosting must meet a minimum specification of 2 vCPUs, 4 GB RAM, and 50 GB storage.	Epic-Deployment	T-Env-003
MR004	The Flutter SDK (latest stable release) and Dart SDK (bundled with Flutter) must be used for development. Development environment must support hot reload and debugging tools.		T-Tool-001
MR005	Android Studio and Xcode must be used for builds and emulators/simulators, including latest stable versions and plugins for Flutter development.		T-Tool-002
MR006	Git and GitHub must be used for version control and collaboration; Git workflow must include feature branches and pull requests.		T-Tool-003
MR007	SQLite must be used for local offline storage.		T-DB-009
MR008	Firebase, AWS, or Azure must be used for authentication, notifications, and backend integration.		T-API-009
MR009	Developer machines must run Windows 10/11 or macOS Monterey+ with all necessary dependencies installed.		T-Env-004
MR010	The application must be deployed to targets running Android 10+ and iOS 15+.		T-Env-005
MR011	A stable broadband connection (≥10 Mbps) must be available for syncing and testing cloud services; latency should not exceed 100ms in normal conditions.		T-Env-006
MR012	All client-server communication must use HTTPS.		T-Security-004
MR013	Developers must test on both Android and iOS environments.		T-Test-001
MR014	The application must comply with Google Play Store and Apple App Store distribution guidelines.		T-Compliance-001

ID	Description	Linked User Stories/Epics	Verification Tests
MR015	Environmental constraints: Development and testing machines must have stable power supply (UPS recommended), controlled room temperature (18–27°C), and low humidity (<60%) to ensure hardware reliability.		T-Env-007
MR016	Performance expectations: Developer workstations and servers must sustain concurrent builds and testing workflows without degradation; response times for cloud API calls must meet SLA targets.		T-Perf-001

2.3. Implementation

2.3.1. Functionality Flow Diagram

Before diving into the implementation fragments, it is useful to visualize the overall functionality flow of the Professional Portfolio system. The following diagram shows how the main user interactions—such as authentication, profile management, swiping, and messaging—trigger backend processes and data persistence operations.

This visualization complements the subsequent implementation fragments by illustrating how user-facing features are realized through coordinated service logic and database operations, ensuring alignment between system behavior and architectural design.

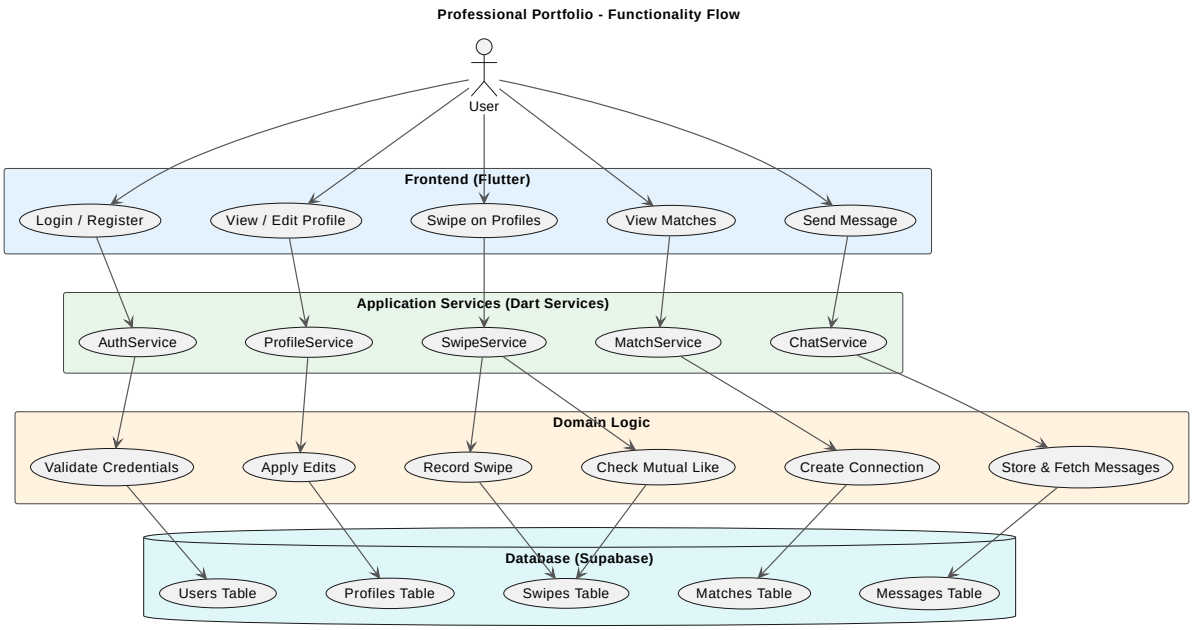


Figure 2.3.1. - 1. Functionality Flow Diagram illustrating the end-to-end process from user interactions to backend persistence.

The diagram clarifies how user interactions are processed across the system's layers. For example,

when a user performs a “Like” action, the event triggers the `SwipeService`, which validates and records the swipe, checks for mutual matches, and—if both users liked each other—creates a new `Connection` entity in the `Matches` table.

This flow contextualizes the domain-centric method signatures presented in the next section, showing how each function contributes to the overall behavior of the system.

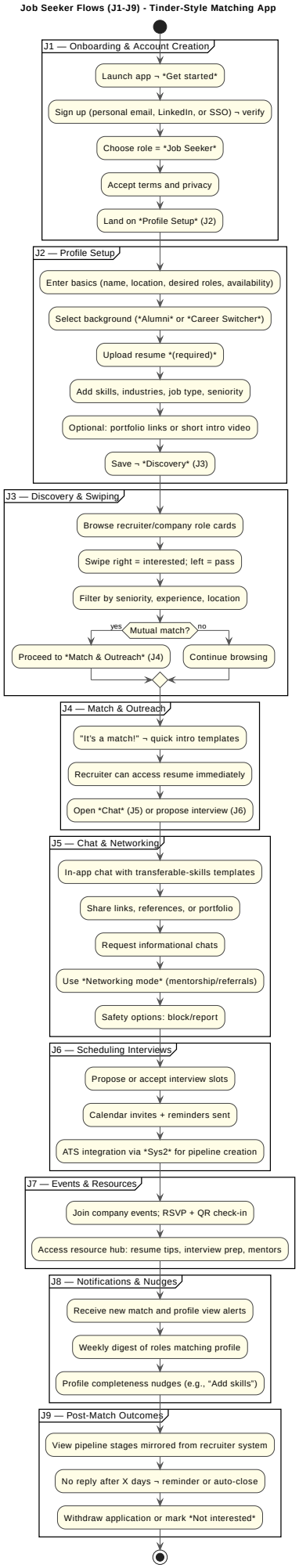

Job Seeker Flows (J1-J9) - Tinder-Style Matching App
J1 — Onboarding & Account Creation
Launch app ¬ *Get started*
Sign up (personal email, LinkedIn, or SSO) ¬ verify
Choose role = *Job Seeker*
Accept terms and privacy
Land on *Profile Setup* (J2)
J2 — Profile Setup
Enter basics (name, location, desired roles, availability)
Select background (*Alumni* or *Career Switcher*)
Upload resume *(required)*
Add skills, industries, job type, seniority
Optional: portfolio links or short intro video
Save ¬ *Discovery* (J3)
J3 — Discovery & Swiping
Browse recruiter/company role cards
Swipe right = interested; left = pass
Filter by seniority, experience, location
Mutual match?
yes
no
Proceed to *Match & Outreach* (J4)
Continue browsing
J4 — Match & Outreach
"It's a match!" ¬ quick intro templates
Recruiter can access resume immediately
Open *Chat* (J5) or propose interview (J6)
J5 — Chat & Networking
In-app chat with transferable-skills templates
Share links, references, or portfolio
Request informational chats
Use *Networking mode* (mentorship/referrals)
Safety options: block/report
J6 — Scheduling Interviews
Propose or accept interview slots
Calendar invites + reminders sent
ATS integration via *Sys2* for pipeline creation
J7 — Events & Resources
Join company events; RSVP + QR check-in
Access resource hub: resume tips, interview prep, mentors
J8 — Notifications & Nudges
Receive new match and profile view alerts
Weekly digest of roles matching profile
Profile completeness nudges (e.g., "Add skills")
J9 — Post-Match Outcomes
View pipeline stages mirrored from recruiter system
No reply after X days ¬ reminder or auto-close
Withdraw application or mark *Not interested*

Figure 2.3.1 - 2. Jobseeker flows diagram showing jobseeker interactions (search, view jobs, apply, save, follow-up) and how those actions map to backend services such as profile updates, application repository writes, and notification triggers.

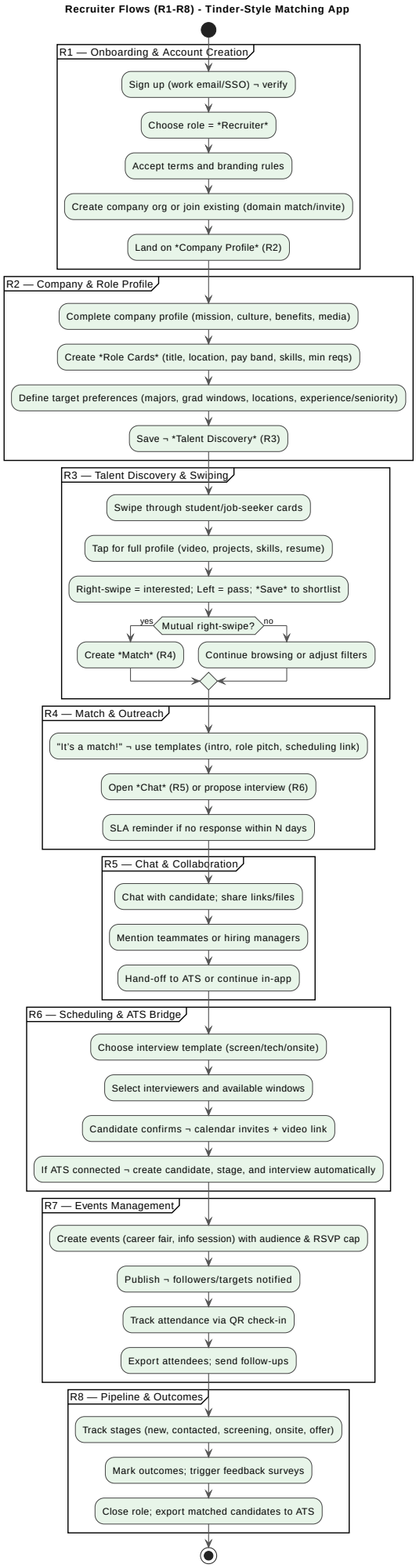

Recruiter Flows (R1-R8) - Tinder-Style Matching App
R1 — Onboarding & Account Creation
Sign up (work email/SSO) ¬ verify
Choose role = *Recruiter*
Accept terms and branding rules
Create company org or join existing (domain match/invite)
Land on *Company Profile* (R2)
R2 — Company & Role Profile
Complete company profile (mission, culture, benefits, media)
Create *Role Cards* (title, location, pay band, skills, min reqs)
Define target preferences (majors, grad windows, locations, experience/seniority)
Save ¬ *Talent Discovery* (R3)
R3 — Talent Discovery & Swiping
Swipe through student/job-seeker cards
Tap for full profile (video, projects, skills, resume)
Right-swipe = interested; Left = pass; *Save* to shortlist
Mutual right-swipe?
yes
no
Create *Match* (R4)
Continue browsing or adjust filters
R4 — Match & Outreach
"It's a match!" ¬ use templates (intro, role pitch, scheduling link)
Open *Chat* (R5) or propose interview (R6)
SLA reminder if no response within N days
R5 — Chat & Collaboration
Chat with candidate; share links/files
Mention teammates or hiring managers
Hand-off to ATS or continue in-app
R6 — Scheduling & ATS Bridge
Choose interview template (screen/tech/onsite)
Select interviewers and available windows
Candidate confirms ¬ calendar invites + video link
If ATS connected ¬ create candidate, stage, and interview automatically
R7 — Events Management
Create events (career fair, info session) with audience & RSVP cap
Publish ¬ followers/targets notified
Track attendance via QR check-in
Export attendees; send follow-ups
R8 — Pipeline & Outcomes
Track stages (new, contacted, screening, onsite, offer)
Mark outcomes; trigger feedback surveys
Close role; export matched candidates to ATS

Figure 2.3.1 - 3. Recruiter flows diagram illustrating recruiter workflows (post job, screen applicants, schedule interviews, take notes), ATS interactions, prescreen operations, and note persistence in repositories.

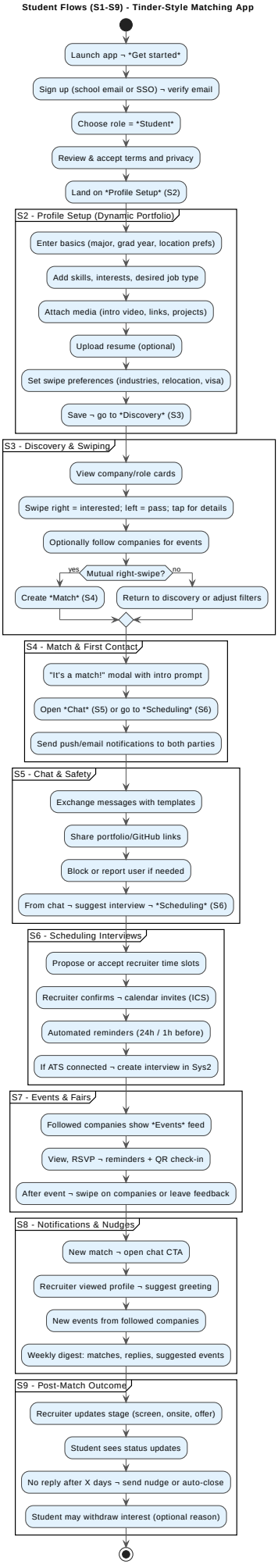

Student Flows (S1-S9) - Tinder-Style Matching App
Launch app ¬ *Get started*
Sign up (school email or SSO) ¬ verify email
Choose role = *Student*
Review & accept terms and privacy
Land on *Profile Setup* (S2)
S2 - Profile Setup (Dynamic Portfolio)
Enter basics (major, grad year, location prefs)
Add skills, interests, desired job type
Attach media (intro video, links, projects)
Upload resume (optional)
Set swipe preferences (industries, relocation, visa)
Save ¬ go to *Discovery* (S3)
S3 - Discovery & Swiping
View company/role cards
Swipe right = interested; left = pass; tap for details
Optionally follow companies for events
Mutual right-swipe?
yes
no
Create *Match* (S4)
Return to discovery or adjust filters
S4 - Match & First Contact
"It's a match!" modal with intro prompt
Open *Chat* (S5) or go to *Scheduling* (S6)
Send push/email notifications to both parties
S5 - Chat & Safety
Exchange messages with templates
Share portfolio/GitHub links
Block or report user if needed
From chat ¬ suggest interview ¬ *Scheduling* (S6)
S6 - Scheduling Interviews
Propose or accept recruiter time slots
Recruiter confirms ¬ calendar invites (ICS)
Automated reminders (24h / 1h before)
If ATS connected ¬ create interview in Sys2
S7 - Events & Fairs
Followed companies show *Events* feed
View, RSVP ¬ reminders + QR check-in
After event ¬ swipe on companies or leave feedback
S8 - Notifications & Nudges
New match ¬ open chat CTA
Recruiter viewed profile ¬ suggest greeting
New events from followed companies
Weekly digest: matches, replies, suggested events
S9 - Post-Match Outcome
Recruiter updates stage (screen, onsite, offer)
Student sees status updates
No reply after X days ¬ send nudge or auto-close
Student may withdraw interest (optional reason)

Figure 2.3.1 - 4. Student flows diagram capturing student-specific steps including event check-in, resume tailoring, booth pitch, informal networking, and multichannel application submission, mapped to system events and follow-up operations.

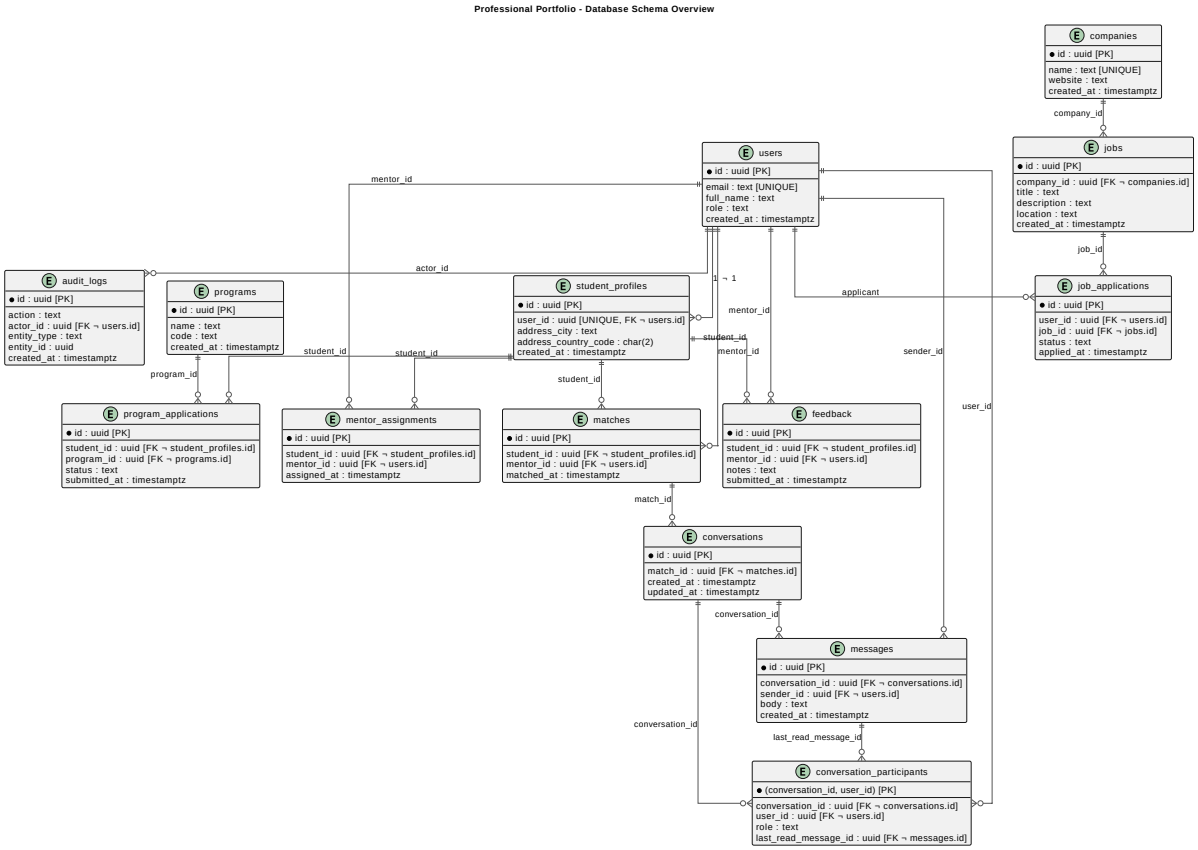


Figure 2.3.1 - 5. Database schema diagram (ER) showing core domain tables — users, student_profiles, programs, program_applications, mentor_assignments, matches, conversations, messages, conversation_participants, feedback, audit_logs, companies, jobs, and job_applications — with primary keys, foreign keys, relationships, and key indices used for deduplication, matching, and conversation threading.

2.3.2. Selected Fragments of the Implementation

The implementation fragments presented here illustrate how the proposed system realizes its core concepts—profiles, swiping, matching, messaging, and event participation—within the domain of student-recruiter interactions. They are not exhaustive; instead, they demonstrate how selected components are translated into concrete structures and operations while maintaining a **clear separation between domain logic and UI representation**. This separation ensures that domain functions (business rules, state changes, eligibility checks) are distinct from presentation widgets (ProfileCard) that render the data.

Domain-Centric Function Signatures

In Flutter/Dart, domain contracts are expressed through method signatures and service interfaces. These contracts define operations on the **Profile** entity and related aggregates, independent of UI concerns. The following functions operate solely on domain concepts:

```
// Fetch the next domain entity for a user's discovery flow
```

```
Future<Profile?> getNextProfile(String userId);

// Record a domain evaluation (Interested / NotInterested)
Future<Result<void>> processSwipe(String userId, String profileId, SwipeDirection
direction);

// Check for reciprocal interest to create a Match
Future<Match?> checkForMatch(String userId, String profileId);

// Create a domain-level Connection entity after a Match
Future<Connection> establishConnection(Match match);

// Send a message within a domain Connection
Future<Result<Message>> sendMessage(String connectionId, String userId, String
content);
```

Each function implements a specific business rule:

- `processSwipe` persists a user's intent to Like or Pass a profile while ensuring it aligns with domain validation rules.
- `checkForMatch` enforces the domain rule that a Match only occurs when both parties express interest.
- `establishConnection` creates the communication context at the domain level, storing history and metadata.

User Interface Complement

The ProfileCard widget represents the presentation layer. It renders information from the Profile domain entity but does not encapsulate domain rules. Interaction events (onLike, onPass) are passed as callbacks that invoke the corresponding domain functions, ensuring that UI logic is separated from business logic.

```
class ProfileCard extends StatelessWidget {
  final Profile profile;
  final VoidCallback onLike;
  final VoidCallback onPass;
  final VoidCallback onMoreInfo;

  const ProfileCard({
    required this.profile,
    required this.onLike,
    required this.onPass,
    required this.onMoreInfo,
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return Card(
```

```
      margin: const EdgeInsets.all(12),
      child: Column(
        children: [
          Text(profile.name, style: Theme.of(context).textTheme.headline6),
          Text(profile.details),
          Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              IconButton(icon: Icon(Icons.close), onPressed: onPass),
              IconButton(icon: Icon(Icons.favorite), onPressed: onLike),
              IconButton(icon: Icon(Icons.info), onPressed: onMoreInfo),
            ],
          )
        ],
      ),
    );
  }
}
```

2.3.3. Visual Representation of the Implementation

This section presents the **actual UI visuals** of the application, showing how domain concepts are represented to the user. The images illustrate the design of ProfileCards, swiping interactions, match notifications, messaging screens, and event participation views.

Chapter 3. Analytic Part

3.1. Concept Analysis

This section refines the raw material from the "Domain Rough Sketch" into a set of domain concepts, abstractions, and relationships, following the guidelines for concept analysis. The process began by reviewing unprocessed notes and quotes, then abstracting recurring ideas into general concepts, clarifying their meaning, and documenting how they relate. This analysis provides a foundation for requirements and system design.

3.1.1. Method

We systematically reviewed the rough sketch, highlighting recurring themes, terms, and pain points. For each, we considered whether it represented a domain concept, event, or behavior, and how it might generalize beyond the specific example. Where terms were ambiguous or used differently by stakeholders, we clarified or resolved them for consistency.

3.1.2. Key Concepts Identified

Queue/Waiting Time: Multiple students mentioned long waits at job fairs. We abstract this as the concept of a "queue," representing bottlenecks in event based interactions.

Profile: resumes, portfolios, and company descriptions are generalized as "profiles", structured representations of actors in the domain. We distinguish between candidate and recruiter profiles.

Match and Connection: Mutual interest is abstracted as a "Match." If acted upon, this becomes a "connection", a persistent relationship enabling further interaction.

Discovery Feed: Reviewing multiple candidates or companies is generalized as a "discovery feed," the set of profiles available for evaluation.

Event: Job fairs, meetups, and interviews are all instances of "events," structuring interactions in time and space.

Note Taking: Recruiter’s use of spreadsheets and notes is abstracted as "record keeping," essential for memory and making decisions in high volume interactions.

Application: Submitting interest through various channels is generalized as an "application," linking candidates to opportunities.

3.1.3. Clarifications and Resolutions

- The term "candidate" is used broadly to mean any job seeker, regardless of experience.
- "Recruiter" refers specifically to the human actor representing an employer, not the company itself.
- "Profile card" is distinguished from "profile" as the interactive, condensed representation used in the discovery process.

- "Match" is an event, while "connection" is a state that persists after a Match.

3.1.4. Relationships and Abstractions

- Candidates build and share their profiles, skills, and qualifications with recruiters.
- Recruiters evaluate candidates based on these profiles, often within the context of an event.
- When mutual interest is signaled, a Match is formed, which can become a connection.
- Connections enable further actions, such as messaging or scheduling interviews.
- Events provide the environment where many of these interactions are initiated.
- Record keeping and note taking support making decisions and memory throughout the process.

3.1.5. Reasoning and Process

This analysis was grounded in the raw observations and quotes from the rough sketch, ensuring that abstractions are traceable to real domain phenomena. Ambiguities were resolved by referencing both stakeholder language and the needs of the requirements phase. By documenting this process, we ensure that the resulting vocabulary is both consistent and shared across the team.

Through this analysis, the scattered ideas from the descriptive phase are distilled into a structured vocabulary. These concepts and relationships now form a shared foundation for the requirements and system design phases that follow.

3.2. Validation and Verification

The purpose of this section is to document how the team will validate and verify the domain concepts, requirements, and design decisions.

- **Validation**: ensuring that what we documented reflects the real-world domain (student–recruiter interactions, job fairs, portfolio showcases, and online hiring practices).
- **Verification**: ensuring that the documentation is internally consistent, complete, and aligned with the project's goals.

3.2.1. Validation Strategy

Validation activities focus on ensuring that our assumptions, workflows, and requirements accurately reflect real-world recruiting, mentoring, and hiring practices. Rather than relying solely on abstract flows, this section grounds validation through concrete, scenario-based walkthroughs involving realistic personas and contexts.

Literature and Online References Foundational validation draws from established data and professional frameworks:

- The **National Association of Colleges and Employers (NACE)** reports that **over 80% of employers evaluate student portfolios or online profiles as part of their screening process**.
- **LinkedIn's Global Talent Trends** emphasize that **skills, projects, and practical experiences** now outweigh traditional credentials.

- **University career centers** (e.g., UC Berkeley, Purdue, UPRM) encourage students to include **artifacts, capstone projects, and self-assessments** in their portfolios, aligning with our design decisions for portfolio sections and recruiter search filters.

Scenario Walkthroughs (Concrete Personas for Validation) Early walkthroughs were initially too abstract, which limited the kind of stakeholder insights we could extract. The updated validation strategy introduces **detailed, narrative-driven personas** that simulate realistic interactions between users and the system. These narratives promote richer discussions with stakeholders by presenting grounded, relatable contexts.

Scenario 1: Abdul (Student) and Corey (Recruiter) — Capstone Connection

Abdul is a 5th-year Mechanical Engineering student finishing his final semester. He is taking **Thermodynamics**, **Pottery II**, and the **Capstone Design** course. His capstone project, **"Predicting Beer Pleasantness from Brewing Parameters,"** leverages data modeling and process optimization — experience relevant to industrial process control.

Corey is a recruiter for **Lilly**, seeking candidates for a process control engineering position. Using the Professional Portfolio platform, Corey filters candidates by "data analysis" and "process modeling." Abdul's profile appears, but his capstone project is missing. Abdul updates his profile, adding a brief description and project artifacts. Once saved, the system automatically updates search indices. Corey's filter now surfaces his profile again, and she swipes right to indicate interest.

Validation Points: - Tests whether **profile updates** propagate correctly to recruiter searches. - Validates **event-driven refresh triggers** in recruiter dashboards. - Checks whether **project tagging and search weighting** align with recruiter expectations. - Reveals potential enhancement: recruiter alerts for updated profiles that match their saved filters.

Scenario 2: Aisha (Mentor) and Daniel (Student) — Mentorship Match Validation

Aisha is a volunteer mentor from the tech industry who joined the platform through a university partnership. She specifies her domain as "data science and product analytics." Daniel, a computer engineering student, signs up for mentoring. He lists his interests as "AI systems," "data visualization," and "career readiness."

The system's mentor matching service pairs them based on overlapping interests. The match occurs because the platform organizes expertise, skills and interests into a shared parent domain "Data Science",which includes subdomains such as AI systems and data visualization. Daniel's interests fall under these subdomains, allowing the matching algorithm to recognize a conceptual overlap between his interest and Aisha's data-science expertise, making her a suitable mentor.

During the first meeting, Daniel mentions wanting to add his internship project portfolio. Aisha advises him to include visualizations of his code metrics. Later, Daniel updates his portfolio accordingly.

Validation Points: - Confirms **mentor matching criteria** reflect relevant skill intersections. - Tests **feedback loops** where mentors influence students' portfolio enrichment. - Surfaces design opportunities for **mentor notes or endorsement badges** to appear on student profiles.

Scenario 3: Rafael (Recruiter) — Shortlisting and Feedback Integration

Rafael is a recruiter from a mid-sized design firm reviewing candidates for UX positions. He uses the filtering tool to shortlist candidates by “portfolio completeness > 80%” and “projects with client collaboration.” He finds Emma’s profile and likes her work but notices missing context for one project. He leaves structured feedback through the recruiter interface, suggesting more detail on design impact.

A few days later, Emma edits her project entry, expanding her case study. The system notifies Rafael that an update was made. This re-engagement loop allows him to reopen her profile and mark it as “ready for review.”

Validation Points: - Tests **feedback submission and notification logic** between recruiters and students. - Validates **recruiter engagement retention** through automatic follow-ups. - Identifies possible feature expansion for **feedback analytics dashboards**.

Scenario 4: Lucia (Career Advisor) — Oversight and Data Consistency

Lucia, a career center advisor, monitors how active recruiters interact with student portfolios. She notices inconsistent data entry (some portfolios lack tags). She requests that the platform provide a “data completeness report” per department. The request leads to the idea of integrating a **Portfolio Health Metric** — a completeness score visible to both students and advisors.

Validation Points: - Ensures **role-based access control** allows advisors to view, not alter, student data. - Encourages new features that support **data quality monitoring** for institutional users.

- **Stakeholder Proxies and Next Steps** Until direct recruiter and mentor interviews begin, **career center advisors** and **industry-aligned professors** serve as proxies for validation. These personas, rooted in observed academic and recruitment contexts, will evolve as we collect field data from pilot users. Future steps include:
  - Conducting **cognitive walkthroughs** with recruiters and students to validate UI intuitiveness.
  - Running **A/B tests** on search filters and project visibility ranking.
  - Integrating qualitative feedback into the next iteration of **matching and notification workflows**.

3.2.2. Verification Strategy

- **Peer Reviews (Planned)** Each section of the documentation (terminology, requirements, narrative) will be reviewed by a different team member. Reviews confirm consistent terminology, measurable requirements, and proper linkage to design elements. Early reviews revealed ambiguities between the terms “candidate” and “student,” which were clarified.
- **Checklists** A milestone-based checklist will be used to explicitly verify each requirement and trace it to design and implementation elements:
  - Every requirement uses only defined terms.
  - Requirements map to at least one domain concept and at least one corresponding design element.

    - No contradictions exist between terminology, requirements, and design.
    - Requirements are measurable and testable (planned for Milestone 2).
    - Step-by-step verification actions are defined for each requirement where applicable (e.g., functional test procedures, walkthroughs).
- **Traceability Matrix (Planned)** A lightweight traceability matrix will be maintained to link requirements to goals, terminology, verification methods, and design elements. This ensures all verification is explicit and measurable:
    - **Requirements → Goals** (ensuring each requirement supports at least one documented goal).
    - **Requirements → Terminology** (ensuring consistent and correct terminology).
    - **Requirements → Verification Method** (explicitly showing how each requirement will be verified, including specific tests or procedures).
    - **Requirements → Design / Implementation** (ensuring traceability from requirement to the actual design and implementation artifacts).
- **Walkthroughs of Documentation** Reader walkthroughs will be simulated by having a team member act as an external reviewer to check that each requirement traces back to definitions, models, and implementation elements. This identifies undefined or ambiguous terms and confirms measurable verification coverage.

3.3. Architecture

3.3.1. ProfileFactory Design (Lecture Topic Task)

Creating a `StudentProfile` or `RecruiterProfile` are profiles that represent aggregate roots in the domain, and they need to start in a valid state from the moment they are created. Because of this, using a factory for profile creation helps us keep all the rules and requirements in one place and prevents invalid profiles from entering the system.

Purpose of the ProfileFactory: The goal of the ProfileFactory is to make profile creation consistent and intentional. Instead of having different parts of the system build profiles in different ways, the factory ensures that every new profile is created with the required data and follows the domain rules. This avoids scattered initialization logic and keeps the model cleaner and easier to maintain.

Profile Invariants: When a profile is created, the following conditions must always be true:

- Every profile must have a unique, immutable UUID. This identity is assigned when the profile is created and cannot change later.
- A profile must start with at least one resume or introductory artifact. This prevents “empty” profiles that cannot participate in matching or recruiter searches.
- A profile must include basic preferences or metadata. These initial settings help guide matching and filtering from the beginning.
- A profile cannot be created for an invalid or incomplete student or recruiter. Any required fields or checks need to be handled inside the factory.

By enforcing these invariants inside the factory, we avoid inconsistent or partially formed profile objects entering the system.

Factory Responsibilities: The `ProfileFactory` is responsible for:

- Generating the UUID for the new profile
- Checking that all required fields are present
- Attaching the initial resume or introductory file
- Initializing skills, metadata, and preferences with valid starting values
- Returning a fully built `StudentProfile` or `RecruiterProfile` object

This keeps the creation logic in one place and makes the model easier to understand and evolve as the system grows.

Conceptual Example (Illustrative Only): The example below shows how the factory might be used. This is not an implementation, just a conceptual illustration of how creation would work in the domain:

```
profile = ProfileFactory.createStudentProfile(
    studentId,
    initialResume,
    defaultPreferences
)
```

This highlights how creation is meant to go through a single, well defined operation rather than ad hoc UI or infrastructure code.

Relation to the Profile Aggregate: The Profile aggregate includes elements such as resume data, skills, preferences, and other metadata. The factory is responsible for assembling these parts correctly at creation time. After creation, the aggregate can evolve normally, but the initial construction should always come through the factory so the domain rules are respected from the beginning.

Using a factory in this way makes the model clearer, keeps invalid states out of the system, and aligns with Domain Driven Design by giving profile creation a dedicated and meaningful place in the overall architecture.

3.3.2. Conceptual Contours Refactor (Lecture Topic Task)

This focuses on clarifying the “conceptual contours” in the current design essentially, identifying where parts of the model have responsibilities that do not naturally belong together. During the review of the MatchingService and ProfileService, a few areas stood out where the logic would benefit from being grouped more intentionally. The goal is not to implement changes, but to surface where the design can be clearer and more cohesive.

Importance: When classes blend together operational logic and policy rules, it becomes harder to understand what each layer is responsible for. This can make the system feel more complicated

than it really is, especially when adding new matching criteria or adjusting profile related rules. By organizing the model according to its natural boundaries, the design becomes easier to reason about and maintain.

What was identified: Two specific issues appeared while reviewing the existing documentation and diagrams:

- Some policy decisions inside the **MatchingService** (like how to weigh certain profile attributes) do not belong in the same place as the core matching workflow.
- The **ProfileService** performs certain data preparation steps that are used specifically for evaluating matches. This mixes profile management with match related decision making.

These do not break the design, but they create friction when the team tries to extend or refine the matching logic.

Proposed Adjustment: To respect the natural boundaries of the domain, the policy related logic should be separated into a dedicated class `MatchingPolicy`. This class would group the configurable or adjustable rules used when evaluating compatibility between a student and a recruiter. A policy class makes the intention clearer: MatchingService handles matching, while MatchingPolicy handles the “rules of how matching should behave.”

This separation aligns with the conceptual contours described in class: a model becomes easier to evolve when each part reflects one clear idea. After extracting policy oriented logic into its own class, both MatchingService and ProfileService would be slimmer and more focused on their specific concerns.

Updated Diagram (Conceptual Overview): The following diagram illustrates how responsibilities can be reorganized to reflect these clearer contours:

```
MatchingService
     ├── runMatchingWorkflow()
     ├── evaluateCandidate()
     └── delegates to MatchingPolicy → applyRules()

MatchingPolicy
     ├── weightSkills()
     ├── checkGPAThreshold()
     └── evaluatePreferences()

ProfileService
     ├── manageProfileData()
     ├── updateSkills()
     └── prepareProfileAttributes()
```

3.3.3. Conservation Principle in Recruiter Attention (Lecture Topic Task)

This introduces the idea of recruiter attention being treated as a limited resource within the domain. During real recruiting interactions especially in university events and portfolio reviews recruiters often describe having only a certain amount of attention or “capacity” to spend on

students. Because of this, modeling attention as a conserved quantity helps the matching process reflect more realistic decision making.

Importance: Right now, the documentation treats matching as if recruiters can evaluate an unlimited number of students equally. In reality, recruiters have constraints: time, focus, and workload. When attention is modeled explicitly, it becomes clearer how the system should behave when a recruiter is overloaded or when too many candidate interactions are happening at once. This refinement aligns the model more closely with real behavior and makes matching policies easier to justify and extend.

Concept Explanation: Under the conservation principle, each recruiter starts with a fixed amount of "attention points." As matches form or as students are evaluated, the recruiter allocates portions of these points. The important part is that the total allocation cannot exceed the fixed amount they have available. This keeps the model consistent and avoids unrealistic scenarios where a recruiter engages deeply with an unlimited number of candidates.

This does not require implementing the logic. Instead, the goal here is to document the concept clearly so that future architectural decisions can rely on a better conceptual foundation.

Conceptual Formula: To show the idea clearly, we include a lightweight model of how attention could be expressed:

```
totalAttention = 100  (example baseline)

sum(attentionAllocatedToStudents) ≤ totalAttention

If allocation reaches the limit:
    → matching requests should be queued, deprioritized, or require policy
intervention
```

Relation: Modeling attention as a conserved resource helps explain why certain recruiter interactions should be limited or prioritized. It also highlights a realistic scenario frequently mentioned in class discussions: recruiters cannot meaningfully engage with infinite candidates, and the system should reflect that constraint.

Documenting this principle strengthens the domain model by:

- Making recruiter constraints explicit instead of implied
- Offering a clearer foundation for future matching policies
- Helping the team reason about prioritization and overload
- Supporting deeper discussions about fairness and load balancing in matching

Even without code changes, capturing this idea in the architecture provides a more grounded way to think about recruiter engagement and gives the next milestones a stronger model to build on.

Chapter 4. Specific Class Topics

4.1. Communication and the Use of Language

4.1.1. Knowledge Crunching for Recruiter Needs

NOTE	Objective: Understand what recruiters actually need from messaging and notifications to guide near-term design/implementation.

One-Page Interview Guide

- Which events are immediate vs batch later? Give concrete examples.
- During quiet hours, what must still break through (by role/stage)?
- Preferred channel per event (push, in-app, email, SMS opt-in) and why.
- Timing windows: reminders (T-24h/T-2h), follow-ups after 48–72h silence, offer deadlines.
- Inbox filters you must have (Interview Today, Needs Feedback, New Priority Match, At-Risk/Idle X days).
- Triage: labels, urgency flags, snooze/mute rules.
- Exact deep-link target on open (thread top, interview card, offer card, note anchor).
- Receipts/“seen” policy (opt-in? per thread/org?). Etiquette concerns.
- Internal mentions: who gets notified and when (@roles, offer steps).
- Automatic follow-ups (first/second nudge cadence, templates).
- Compliance/consent: time zones, opt-in for SMS/email, do-not-disturb, retention.
- One thing current tools get wrong about notifications—and how to fix it.

Interviews

Interview 01: Mid-market Tech Recruiter · Date: 2025-10-05 · Mode: phone (28 min) · Region: urban + suburban sites · Anonymized

Answer 1: Immediate vs batch. *Immediate:* same-day interviews, <24h reschedules, offer steps. *Batch:* new applicants, minor status changes.

Answer 2: Quiet hours. 7pm–7am local; exceptions: <24h reschedule, offer accept/expire.

Answer 3: Channels. Urgent → push + in-app (SMS if candidate opted-in); routine → in-app + email digest.

Answer 4: Timing. Reminders T-24h/T-2h; feedback prompt 2h post-interview; no-reply nudges at 48h/96h.

Answer 5: Filters. Interview Today, Needs Feedback, New Priority Match, Idle 3+ days.

Answer 6: Triage. Urgent flag; snooze until 08:00; mute threads after closure.

Answer 7: links. Open the interview card pinned inside the thread.

Answer 8: Receipts. Delivery timestamp OK; “seen” off by default (adds pressure).

Answer 9: Internal. Hiring Manager on interview feedback; Finance only on offer.

Answer 10: Automation. “Thanks—feedback soon” to candidates; 48/96h nudges to keep momentum.

Answer 11: Compliance. Respect candidate time zone; store per-channel consent; honor DND.

Answer 12: tools. Too many pings; batch low-signal changes and keep deep links precise.

Structured Notes (Interview 1):

Theme	Signals / Details
Urgent	Interview day, reschedule<24h, offer
Batch	New applicant, non-critical status change
Policies	Per-thread quiet hours + role overrides
Navigation	Deep link → interview/offer card
Etiquette	Delivery receipts over “seen”
Automation	48/96h nudges; 2h feedback prompt

Interview 2: Agency Recruiter (multi-timezone) · Date: 2025-10-06 · Mode: video (28 min) · Region: urban + suburban sites · Anonymized

Answer 1: Immediate vs batch. Immediate: offer steps, day-of schedule changes. Batch: sourcing/status updates.

Answer 2: Quiet hours. After 7pm local; override only for <24h reschedules/offer deadlines.

Answer 3: Channels. Urgent → push; day-of SMS if opted-in; daily email digest for summaries.

Answer 4: Timing. T-24h/T-2h reminders; auto re-invite at 72h no-reply.

Answer 5: Filters. Interview Today, Offer Pending, At-Risk (no response 96h).

Answer 6: Triage. Labels: Urgent / Follow-up / Waiting; snooze per thread.

Answer 7: Deep links. Interview details with join link and call button.

Answer 8: Receipts. Opt-in per thread; org default is off.

Answer 9: Internal. @Legal/@Finance only during offer; no alerts for routine notes.

Answer 10: Automation. Post-interview “thank you” + 48/96h nudges.

Answer 11: Compliance. Regional DND and recorded consent (SMS/email).

Answer 12: Gap in tools. Alerts are not role-aware; everything notifies everyone.

Structured Notes (Interview 2):

Theme	Signals / Details
Urgent	Reschedule<24h, offer, interview reminder
Batch	Sourcing and stage changes
Overrides	Offer steps ignore quiet hours for recruiter/manager
Deep link	Interview details with join link
Automation	72h re-invite; At-Risk at 96h
Consent	Per-channel opt-ins saved

Synthesis

What recruiters actually need from chat/notifications?

- Urgency rules separate true interrupts (reschedule<24h, offer, day-of interview) from batchable noise (status/sourcing).
- Per-thread quiet hours with role-based overrides for offer & day-of scheduling.
- Channels by event (push/in-app urgent; email digest for low-signal; SMS only with opt-in).
- Reliable deep links to the exact action (Interview card, Offer card, Note anchor).
- Inbox controls (Interview Today, Needs Feedback, New Priority Match, At-Risk/Idle).
- Polite automation (48/96h nudges; feedback prompt 2h; 72h re-invite).
- Etiquette & consent defaults (delivery receipts vs “seen”; stored per-channel consent).
- Outcomes: fewer no-shows, faster time-to-respond, clear audit of what alerted whom and why.

Event Taxonomy & Matrix

Event	Actor	When	Priority	Channel	Deep Link
Interview Reminder	Candidate, Recruiter	T-24h, T-2h	High	Push + In-app	Interview card
Reschedule/Cancel <24h	Both	Instant	Break Quiet Hours	Push + SMS (opt-in)	Interview card (new time)
Offer Extended	Candidate, Recruiter	Instant	High	Push + Email	Offer card
Offer Deadline T-48h	Candidate, Recruiter	T-48h	High	Push + In-app	Offer card (deadline)

Event	Actor	When	Priority	Channel	Deep Link
New Priority Match	Recruiter	Instant	High	Push + In-app	Profile → thread
No Reply 48h/96h	Recruiter	Daily batch	Medium	In-app + badge	Thread with “Nudge”
Internal @Mention	Mentioned role	Instant	Normal	In-app + Email	Note anchor
Doc Request	Candidate	Daily batch	Low	Email	Files tab
Snooze Ends	Recruiter	At wake time	Low	In-app	Thread top

Prioritization

Must

- Policy engine for urgency & quiet hours (thread/role aware)
- Reliable deep links (interview/offer/note anchors)
- RSVP + calendar integration
- Morning digest (07:30 local) for low-signal updates
- Auto-nudge at 48/96h silence
- Internal @mentions with anchors

Should

- Opt-in delivery receipts and SLA badges
- Candidate quiet hours and consent per channel
- Role-based overrides for urgent events

Could

- Stage-aware bundling of notifications
- Intent/sentiment hints in threads

Won’t (now)

- Cross-org notification analytics

Implementable Requirements

R1 — Policy Engine for Urgency & Quiet Hours

```
Feature: Notification policy engine

Scenario: Reschedule within 24 hours breaks quiet hours
  Given the thread has quiet hours from 19:00 to 07:00
  And an interview is rescheduled 12 hours before start
  When the event is processed at 22:00
  Then a HIGH priority push notification is sent
  And the deep link opens the interview card

Scenario: Non-urgent updates during quiet hours are batched
  Given quiet hours are active
  When a status change event occurs
  Then no push is sent
  And the update appears in the 07:30 digest
```

R2 — Interview RSVP + Calendar Deep Link

```
Feature: RSVP and calendar

Scenario: Candidate receives invite
  Given an interview invite is sent
  When the candidate opens the notification
  Then the app shows RSVP buttons
  And an .ics file is available
  And the meeting link is visible
```

R3 — Daily Digest at 07:30 Local

```
Feature: Morning digest

Scenario: Recruiter starts day
  Given there are interviews today and 48h+ unreplied threads
  When it is 07:30 local time
  Then a digest groups interviews, unreplied threads, and new priority matches
  And each item deep links to the exact context
```

R4 — Auto-Nudge After Silence (48h/96h)

```
Feature: Auto follow-up

Scenario: First nudge at 48h
  Given a candidate has not replied for 48 hours
  When automation runs
  Then a polite nudge is posted in the thread
```

```
Scenario: At-risk at 96h
  Given no reply after the first nudge
  When 96 hours elapse
  Then a second nudge is posted
  And the thread is labeled "At Risk"
```

R5 — Thread-Level Mute/Snooze

```
Feature: Snooze thread

Scenario: Snooze until tomorrow 08:00
  Given a recruiter snoozes a thread until 08:00 tomorrow
  Then no notifications are sent for that thread until 08:00
  And a "Snoozed" badge is visible
```

R6 — Internal @Mentions with Anchors

```
Feature: Internal mentions

Scenario: Mention hiring manager
  Given a note mentions the hiring manager
  Then the manager receives a notification
  And opening it jumps to the note anchor in the thread
```

R7 — Delivery Receipts + SLA Badges (opt-in)

```
Feature: Delivery receipts and SLA

Scenario: SLA risk after threshold
  Given the organization SLA is response within 24 hours
  And a message has been delivered for 24 hours without reply
  Then the thread shows "SLA at risk"
```

Traceability — Pain → Event → Requirement:

Pain Point	Event(s)	Requirement(s)
Too many alerts / noise overload	Reschedule<24h, Interview reminders, Offer updates	R1 (Policy Engine), R3 (Morning Digest)
Missed follow-ups / dropped threads	No Reply 48h/96h	R4 (Auto-Nudge After Silence)

Pain Point	Event(s)	Requirement(s)
Wrong screen after tapping notification	All (Interview, Offer, Mention)	R2 (Deep Links + RSVP/Calendar)
Coordination gaps between roles	Internal @Mention	R6 (Internal @Mentions with Anchors)
Pressure from read receipts	Any (messages, updates)	R7 (Delivery Receipts opt-in)
Need for silent focus time	All (Batch, low-priority)	R1 (Policy Engine + Quiet Hours)
Inconsistent candidate updates	Interview reminder, Offer deadline	R2 (RSVP/Calendar), R3 (Digest)

References

LinkedIn. (n.d.). Alerts and notifications. Recruiter Help. https://www.linkedin.com/help/recruiter/topic/a100022

LinkedIn. (n.d.). InMail and Inbox. Recruiter Help. https://www.linkedin.com/help/recruiter/topic/a52

Workable. (n.d.). Scheduling events overview. Workable Help Center. https://help.workable.com/hc/en-us/articles/115013135088-Scheduling-events-overview

Workable. (n.d.). Do candidates receive reminder emails for scheduled events? Workable Help Center. https://help.workable.com/hc/en-us/articles/32871072712855-Do-candidates-receive-reminder-emails-for-scheduled-events

Indeed for Employers. (2025, May 27). How to message candidates on Indeed: A guide for employers. https://www.indeed.com/hire/resources/howtohub/indeed-messaging

Indeed for Employers. (2024, November 4). 6 texting and chatting rules with candidates. https://www.indeed.com/hire/c/info/texting-chatting-rules-with-candidates

04CommunicationAndTheUseOfLanguage.pdf — Ubiquitous Language; aligning terms across interviews, requirements, and data/model.

4.1.2. Minimal vs Enriched UML

In the enriched model the operation of accepting or swiping right to allow communication between recruiter and job seeker from the perspective of the recruiter, with the assumption that job seekers have accepted or “swiped right” a job or program that this recruiter manages/is a mentor of. It is decomposed into the layers that are used in the order they are used by the system. It shows 2 user scenarios, one were the connection is available and access to the database is possible in the moment it is requested or “happy path” and another when the access is not possible at that moment which makes the system look for a backup or cache of the matches that could have been retrieved in a previous fetch and if that also fails it will store the user interaction to update database with the accept flag for the role of the current user on the corresponding user of opposite role.

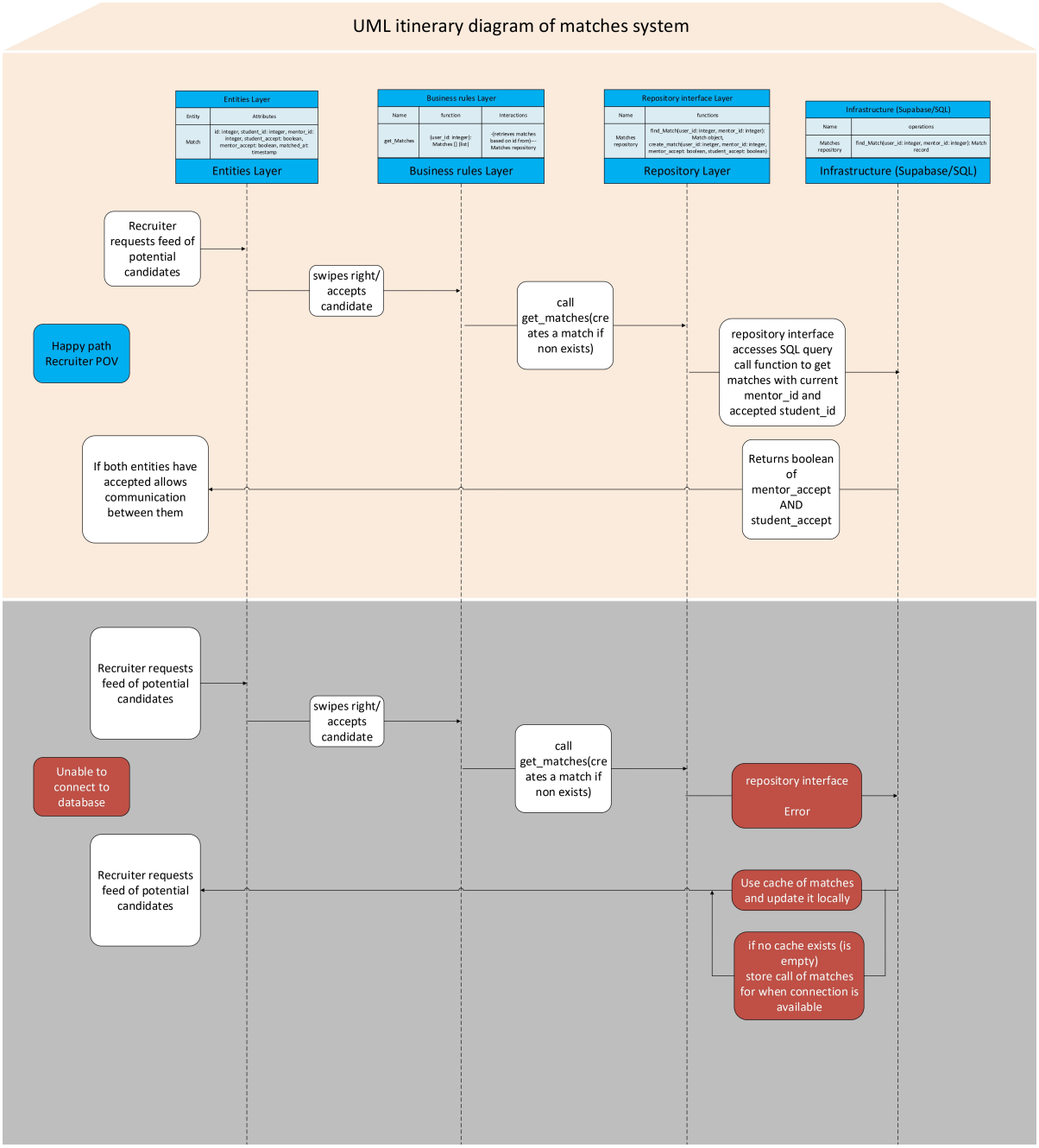


Figure 4.1.2. - 1. Itinerary of recruiter scenario showing the various stages and interactions involved in the recruitment process.

Matches				
Entities Layer		Business rules Layer		
Entity	Attributes	Name	function	Interactions
Match	id: integer, student_id: integer, mentor_id: integer, student_accept: boolean, mentor_accept: boolean, matched_at: timestamp	get_Matches	(user_id: integer): Matches [] (list)	-(retrieves matches based on id from)--- Matches repository
Repository interface Layer		Infrastructure (Supabase/SQL)		
Name	functions	Name	operations	
Matches repository	find_Match(user_id: integer, mentor_id: integer): Match object, create_match(user_id: inetger, mentor_id: integer, mentor_accept: boolean, student_accept: boolean)	Matches repository	find_Match(user_id: integer, mentor_id: integer): Match record	

Figure 4.1.2. - 2. UML diagram showing the entities involved in the matching process between recruiters and students.

4.1.3. Communication and the use of Language: Modeling Out Loud for Swipe Flow

NOTE Objective: To explore and refine the Swipe→Match→Chat domain model using ubiquitous language and the "modeling out loud" technique, ensuring conceptual clarity, identifying domain rules, exposing ambiguities, and aligning the model with the system's requirements.

Modeling out loud: Swipe → Match → Chat

This flow uses a shared language. **A** is the Candidate; **B** is the Recruiter. Both have a public **Profile** and receive recommendations. When A views B's Profile, A can **Swipe**: **Like** (right) or **Pass** (left). That action triggers the flow. The server saves the **A→B Like idempotently** (duplicates count once) and enforces **visibility** (if B is "by match only" or "private," A cannot swipe). No chat occurs yet; **reciprocity** is required—only mutual Likes create a **Match**.

Later, B likes A. The **Matching Service** detects reciprocity, confirms **uniqueness** (only one active Match per pair), and creates `Match(A,B)`. This automatically opens a **1:1 ChatThread**, enabled immediately and seeded with the message **"You matched!"**. The **Notification Service** alerts both sides with retries and backoff; if delivery fails, the notification inbox surfaces the new state on the next session. From that point on, chat is enabled; before that, chat is gated by the Match.

Detours are covered. If either party chooses **Pass**, nothing else happens and the next card appears. A short, server-validated **Undo window** allows reverting the last Swipe; if no Match was formed, the change rolls back cleanly. If visibility changed or a profile was blocked between recommendation and Swipe, the server rejects the attempt with a clear, auditable message.

Repeated Likes beyond the first are ignored due to idempotency.

Safety and exit conditions apply. Either party may **Unmatch** at any time. The ChatThread closes, future messages are rejected, and an audit record is written. If a **Report** is filed, a moderation case opens, and the system may restrict or close the chat according to policy. All actions are auditable, and participants are notified when appropriate.

Testing focuses on observable outcomes. After one Like, intent is saved but no chat appears. After two reciprocal Likes, exactly one `Match(A,B)` exists, one `ChatThread(A,B)` opens, and both parties receive notifications. Order or repetition of Likes must not break idempotency or uniqueness. Visibility rules apply both when showing the card and at Swiped time. Undo only reverts if the action has not yet created shared state (e.g., a Match).

Ubiquitous terms—**Profile, Swipe, Like/Pass, Match, ChatThread, Notification, Undo window, Unmatch, Report, Idempotency, Visibility, Uniqueness, Reciprocity**—should appear in user stories, acceptance criteria, and code (classes, events, endpoints) to keep the model and implementation aligned. When a term becomes ambiguous, the team updates the language and refactors accordingly.

In summary, the minimum scope runs from a Swipe on a valid recommendation to a reciprocal Match, automatic ChatThread opening, and reliable notifications, governed by reciprocity, uniqueness, gated chat, visibility, idempotency, and unmatch/report support, with end-to-end checks. The scope is traceable to Personas and User Stories, Epics, and Features, and is ready for build and review.

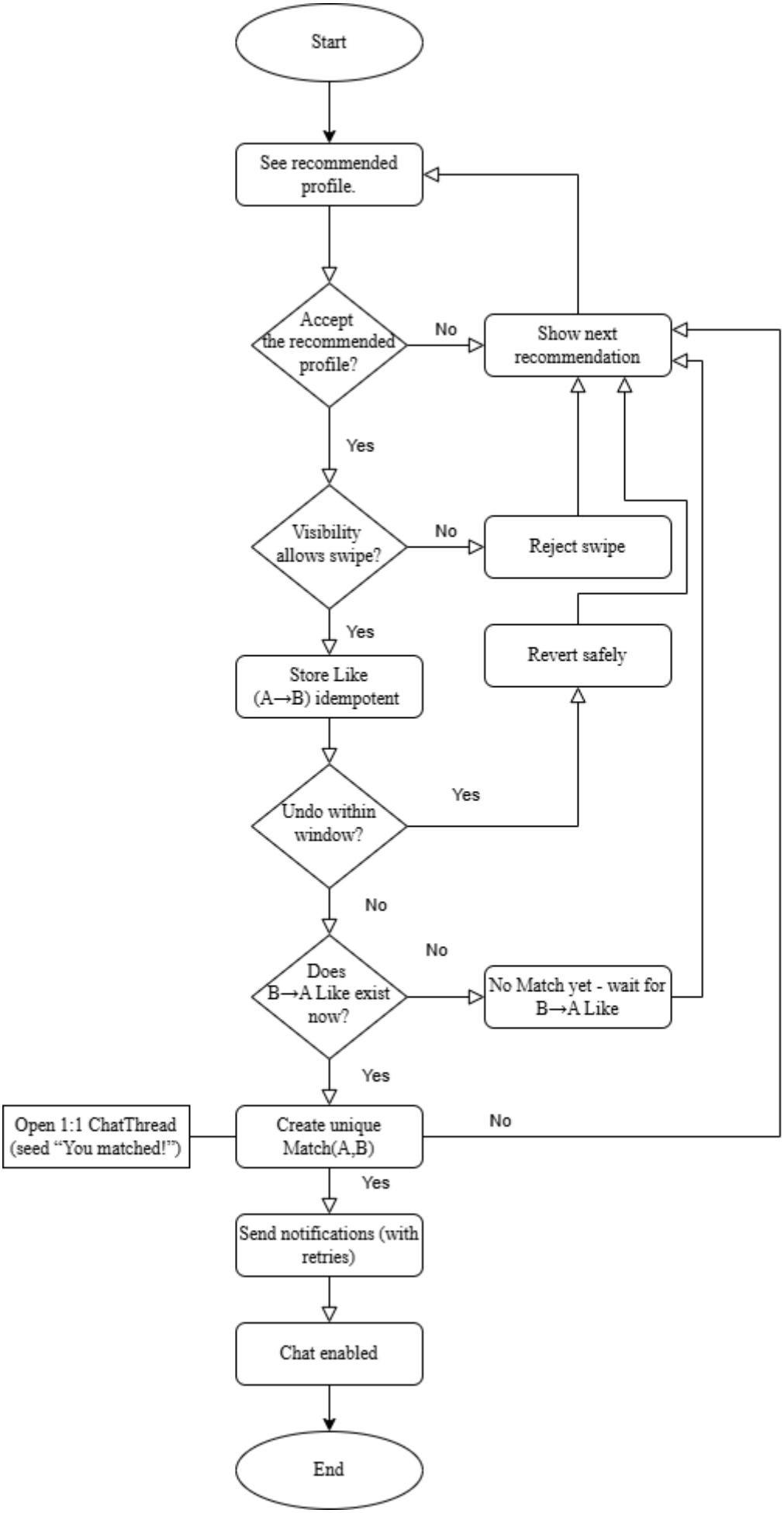


Figure 4.1.3. - 1. Flowchart LTT – Modeling Out Loud for Swipe Flow

This material is guided by sections 1.3 (**Scope, Span, and Synopsis**), 2.1 (**Domain Description**), 2.2 (**Requirements**), and 3.2 (**Validation and Verification**). It draws substantially on the PDF **Communication and the Use of Language** (UPRM, 2020), particularly the guidance on ubiquitous language and “modeling out loud”.

4.1.4. Refactoring Names to Match Ubiquitous Language

NOTE	Objective: Align all names used in the project—classes, files, screens, services, and models—with the Ubiquitous Language established in Milestones 1–3. This task applies the lecture principles from **Communication and the Use of Language**, **Effective Domain Modeling**, and **Making Implicit Concepts Explicit**.

Topic Overview

The lectures emphasize that every concept in a domain-driven design must use a **single, precise term** across the entire system. Ambiguous names, duplicate terms, or UI-driven terminology cause inconsistencies that break communication between developers, designers, and stakeholders.

The Ubiquitous Language is defined by our Milestone 1 Terminology and expanded in Milestones 2 and 3. This task identifies mismatches between that vocabulary and the names currently present in the project structure.

Ubiquitous Language Reference (from M1–M3)

The official terms relevant to naming include:

Profile, ProfileCard Swipe, Like, Pass Match Connection Jobseeker Profile, Recruiter Profile Opening, Application, Requirements, Eligibility Message, Conversation Session Visibility (Public, By Match, Private) Identity (UUID)

These terms guide all naming corrections.

Identification of Naming Inconsistencies

A review of the project’s folder and file structure (as shown in the provided screenshots) revealed the following inconsistencies:

Current Name	Issue	Explanation
`user.dart`	Vague / generic	Domain uses **Profile** to refer to participants; “User” is ambiguous.
`models.dart`	Non-domain, unclear purpose	“Models” does not map to any domain concept.
`conversation.dart`	UI wording	Domain concept corresponds to **MessageThread** or **ConversationThread**.
`recruiter_profile_card.dart`	UI term mixed into model	“Card” belongs to UI; domain uses **ProfileCard** only in discovery flow.

Current Name	Issue	Explanation
`matches.dart`	Plural file for singular domain concept	Domain concept is **Match** (singular, event).
`jobseeker_profile.dart` vs `student_profile.dart`	Terminology conflict	Only one term should represent this domain role.
`mock_data.dart`	Non-domain, generic	Acceptable for mock layer but must not leak into domain naming.
`jobseeker_profile_controller.dart`	UI term acceptable	Should remain clearly UI-only.
`login_controller.dart`	OK, UI-only	No change; matches presentation layer.
`chat_service.dart`	Vague	Domain concept is **MessageService** or **ConversationService**.
`app_router.dart`	Technical name	Acceptable, belongs to infrastructure/UI.

These inconsistencies affect clarity in domain documentation and create hidden mismatches between terminology and implementation.

Proposed Renamed Set (Aligned with Domain Language)

The following replacements maintain meaning while ensuring alignment with official terminology:

Current Name	Recommended Replacement	Rationale
`user.dart`	`profile.dart` or `participant_profile.dart`	Matches domain's Profile concept.
`models.dart`	Remove or rename to `shared_types.dart`	Must not imply domain concept that doesn't exist.
`conversation.dart`	`message_thread.dart`	Matches domain's description of message grouping.
`recruiter_profile_card.dart`	`recruiter_profile_preview.dart`	UI element representing a profile; avoids mixing "card" with domain term **ProfileCard**.
`matches.dart`	`match.dart`	Domain event is singular and atomic.
`jobseeker_profile.dart`	`candidate_profile.dart` OR unify with `student_profile.dart`	Only one canonical term should be used for the candidate role.
`chat_service.dart`	`message_service.dart`	Domain entity is Message, not "chat".
`mock_data.dart`	Leave unchanged, but document as infrastructure-only	Clarifies it is technical, not domain language.

Only names that belong inside the domain layer or affect conceptual clarity are renamed. UI names (controllers, screens, widgets) stay as is—they do not affect domain terminology.

Documentation Updates

To preserve consistency in the Milestone 3 document:

- All references to “user” must be replaced with **Profile** or the appropriate domain role.
- Any mention of “chat” in domain contexts must be replaced with **Message** or **Conversation Thread**.
- Match should appear only in the singular form when referring to the domain event.
- For candidate roles, the terminology must settle on a single label (Student, Jobseeker, or Candidate); the recommended choice is **Candidate**, as it matches Milestone 3 wording.
- The Domain Rules section should avoid UI words such as “card,” except when referring to the discovery **ProfileCard**.

Consistency Verification

Each proposed replacement was validated against:

- Milestone 1 Terminology
- Milestone 2 Narrative and Events
- Milestone 3 Domain Rules (LTT 13)
- The Ubiquitous Language guidelines from lecture materials

The proposed names reinforce domain clarity and remove ambiguity caused by UI or infrastructure terminology.

Traceability to Lecture Concepts

Lecture Concept	Application in Task
Ubiquitous Language	Names unified using consistent domain terms.
Communication & Language	Eliminates ambiguity and improves shared understanding.
Implicit Concepts	Reveals hidden mismatches (e.g., “user” vs “profile”).
Making Domain Concepts Explicit	Each name directly maps to a domain entity or event.

References

- Schütz-Schmuck, Marko. **Communication and the Use of Language** (Lecture Slides 3–6).
- Schütz-Schmuck, Marko. **Effective Domain Modeling** (Lecture Slides 4–7).
- Schütz-Schmuck, Marko. **Making Implicit Concepts Explicit** (Lecture Slides 2–5).

4.2. Isolating the Domain & Expressing Models in Software

4.2.1. Domain Rules in Domain Layer

NOTE	Objective: Ensure all business rules are well established and available for developers to refer to and for potential clients to review.

Matching Policy

For a Match to occur between a Recruiter and a Student, there must first be a similarity between the Student's skillset and a Recruiter's JobOpening. If these skills are close enough, each user will have more chances to see each other in the App, and they can each accept or decline each other. If both users accept each other, a Match occurs, and they may begin interacting with each other via messages.

User Interactions

If a Match occurs, a Recruiter and a Student will begin direct contact through the App's messaging feature. Afterwards, they may arrange meetings and interviews by utilizing external tools to our App, such as Microsoft Teams, Zoom, In Person meetings, etc.

4.2.2. Formal Consolidation of Domain Rules in the Domain Layer

NOTE	Objective: Ensure that all business rules governing Swipe, Match, Visibility, Applications, and Profile interactions are defined strictly within the Domain Layer. This section expands the high-level descriptions included in Milestone 3 by consolidating them into a specialized, authoritative rule set grounded in the course lectures on language, domain isolation, and the clarification of implicit rules.

Topic Overview

The course lectures establish that domain rules must:

- be expressed using the project's Ubiquitous Language,
- exist exclusively inside the Domain Layer,
- avoid dependence on UI gestures, infrastructure concerns, or database-driven logic.

Milestone 3 introduced these rules in narrative form, but they were distributed across several sections (events, user flows, requirements). This section formally unifies them into a cohesive domain rule specification.

Consolidated Domain Rules

Swipe Rules

- A Swipe (Like or Pass) is a domain action tied to Profile identity.

- A Pass prevents rediscovery of the same ProfileCard within a Session.
- A Like expresses interest but does not create a Match on its own.

Match Formation

- A Match occurs only when both Profiles have registered a Like.
- Matches must be unique for each Recruiter–Candidate pair.
- Closed, invalid, or blocked Profiles cannot generate Matches.

Visibility Rules

- Public Profiles appear in recruiter discovery.
- By-Match Profiles become visible only after a Match.
- Private Profiles never appear in recruiter discovery.

Connection Rules

- A Connection is created only from a valid Match.
- Messaging requires an active Connection.
- Connections persist beyond the transient Match event.

Blocking and Reporting

- Blocking prevents future Matches and disables existing Connections.
- Reporting creates a moderation case and restricts interaction.

Undo Clarification

- Undo is a UI-only gesture, not a domain action.
- The Domain Layer recognizes only committed Swipes and Matches.

Eligibility and Requirements

- Eligibility uses the categories: meets, partially meets, does not meet.
- Results must be deterministic and reproducible.

Applications

- Applications bind a Candidate to an Opening.
- Withdrawals preserve historical traceability.
- Applications cannot advance unless requirements are satisfied.

Interview Scheduling

- Allowed only with a valid Match or shortlist.
- Time blocks cannot conflict for either participant.

Offer Rules

- Offers require explicit terms and expiration dates.
- Expired offers cannot be accepted.

Domain Identity

- All entities use immutable UUIDs.
- Identity values cannot change across migrations or updates.

Placement in the Domain Layer

Following the lecture material:

- Swipe, Match, and Visibility rules → Profile, Match, and Connection aggregates
- Blocking and Reporting → Moderation behaviors
- Eligibility, Applications, Interviews, Offers → Opening and Application aggregates
- UUID Identity → Domain identity strategy

No rules appear in UI controllers, animations, gesture handlers, mock data, or database triggers.

Verification and Consistency

The consolidated rule set was validated against:

- Milestone 1 Terminology
- Milestone 2 System Flows
- Milestone 3 Domain Rules
- Course lectures on language, isolation, and explicit domain rule expression

The rules align with the intended system behavior and preserve strict domain isolation.

Traceability to Lecture Concepts

Lecture Concept	Application in This Section
Communication and the Use of Language (Slides 3–6)	Ensures domain rules use precise Ubiquitous Language.
Making Implicit Concepts Explicit (Slides 2–5)	Elevates scattered or implied rules into explicit domain logic.
Isolating the Domain (Slides 3–8)	Separates domain logic from UI gestures and infrastructure concerns.
A Model Expressed in Software (Slides 4–9)	Maps each rule to aggregates and domain events.

References

- Schütz-Schmuck, Marko. **Communication and the Use of Language** (Lecture Slides 3–6).
- Schütz-Schmuck, Marko. **Making Implicit Concepts Explicit** (Lecture Slides 2–5).
- Schütz-Schmuck, Marko. **Isolating the Domain** (Lecture Slides 3–8).
- Schütz-Schmuck, Marko. **A Model Expressed in Software** (Lecture Slides 4–9).

4.2.3. Swipe and Match Logic in the Domain Layer

NOTE	Objective: Define and document the complete domain behavior governing swipe evaluation, interest recording, mutual-interest detection, match creation, and

> connection activation. This task follows the principles from **Effective Domain Modeling**, **Making Implicit Concepts Explicit**, and **Isolating the Domain & Expressing Models in Software**.

Topic Overview

The lecture material emphasizes that business logic must be placed inside the Domain Layer and expressed through domain concepts rather than UI gestures or infrastructure behavior. Swiping, evaluating interest, and forming matches represent core domain transitions. Their rules must be defined using domain terminology and enforced at the boundaries of aggregates such as `Swipe`, `Match`, and `Connection`.

A domain-owned sequence is required to ensure that swiping decisions and match formation are independent of UI concerns. The UI may initiate actions but cannot interpret business rules, compute reciprocity, or construct matches.

Identification of Domain Behaviors

Using terminology from Milestone 1 and behavioral descriptions from Milestone 2 and 3, the system defines:

Swipe — A domain action representing interest (Like) or rejection (Pass). **Match** — An event created only when mutual interest exists. **Connection** — A persistent relationship created as the consequence of a Match. **Session** — The context in which swipes and profile presentation occur.

The task identifies the domain-owned behaviors that were previously implicit or spread across UI descriptions.

Domain-Owned Swipe/Match Sequence

The complete domain sequence is defined as:

1. **Swipe** The actor performs a `Like` or `Pass` on a `ProfileCard`. Swipe is a domain event tied to the actor's identity.
2. **Evaluate Swipe** The Domain Layer records the action and interprets its effect. A `Pass` removes the profile from the current Session. A `Like` records interest but does not trigger additional actions by itself.
3. **Check Mutual Interest** Reciprocity exists only when both parties have previously recorded a `Like` on each other's ProfileCard. The check is expressed as a domain rule: `Like(A,B) AND Like(B,A) → Candidate for Match`.
4. **Create Match** A `Match` is formed only when mutual interest exists. A Match is unique per Recruiter–Student pair and must not duplicate an existing relationship. Matches cannot be created for invalid, blocked, or private profiles.
5. **Trigger Connection** A `Connection` is created as the domain consequence of a valid Match. The Connection persists beyond the transient Match event and serves as the only context for messaging.

This sequence is the domain's source of truth for all swipe and match logic.

Separation From UI Responsibilities

The following are explicitly **not** domain behaviors:

- Swipe gestures, card animations, left/right drag distances
- Undo actions performed in the UI
- Deck navigation or presentation logic
- UI rules about when cards appear or disappear

The domain recognizes only committed domain actions and events.

Domain Constraints and Invariants

The domain enforces the following invariants:

- A Match cannot be formed without mutual interest.
- A Connection cannot be formed without a Match.
- A Pass prevents the same ProfileCard from reappearing within the same Session.
- A Match must be unique and cannot be duplicated.
- Messaging requires an active Connection.
- Blocked users cannot form new Matches or Connections.

These invariants maintain integrity across swiping and matching flows.

Verification and Alignment

The documented sequence was validated against:

- Milestone 1 Terminology
- Milestone 2 Domain Description and Events
- Milestone 3 Domain Rules
- The required system flows (Swipe → Match → Connection)

The rules match the project's official ubiquitous language and remove any dependency on UI conditions.

Traceability to Lecture Concepts

Lecture Concept	Application in Task
Communication and the Use of Language (Slides 3–6)	Swipe, Like, Pass, Match, Connection, Session expressed in official UL.
Making Implicit Concepts Explicit (Slides 2–5)	Mutual interest made explicit as a domain rule.
Isolating the Domain (Slides 3–8)	UI gestures removed; all logic assigned to domain.

Lecture Concept	Application in Task
A Model Expressed in Software (Slides 4–9)	Rules mapped onto aggregates and domain events.

References

- Schütz-Schmuck, Marko. **Communication and the Use of Language** (Lecture Slides 3–6).
- Schütz-Schmuck, Marko. **Making Implicit Concepts Explicit** (Lecture Slides 2–5).
- Schütz-Schmuck, Marko. **Isolating the Domain** (Lecture Slides 3–8).
- Schütz-Schmuck, Marko. **A Model Expressed in Software** (Lecture Slides 4–9).

4.2.4. Modeling the Address as a Value Object

NOTE	Objective: Refactor the `StudentProfile` aggregate to incorporate `Address` as an immutable Value Object, applying the principles from the lectures on domain isolation, value object modeling, and expressing domain concepts through aggregates, factories, and repositories. The goal is to keep domain logic pure, enforce invariants at construction time, and preserve persistence independence through a repository layer.

Topic Overview

The lecture material emphasizes that Value Objects represent concepts defined entirely by their attributes, must be immutable, and must be validated at creation time. Entities, by contrast, are defined by identity and serve as aggregate roots controlling updates to their internal objects. The lectures also highlight the role of repositories and factories in separating domain logic from storage concerns, keeping the domain model independent of external infrastructure.

Application in the Project

In the Professional Portfolio System, the `StudentProfile` aggregate initially scattered raw address fields (`street`, `city`, `postal_code`, etc.), which led to duplicated logic and weak cohesion. Through knowledge-crunching and discussion, this revealed an **implicit concept** — the Address itself. The team made it explicit as a dedicated `Address` Value Object embedded within the `StudentProfile` aggregate root.

The `StudentProfileRepository` mediates between the domain and Supabase, exposing methods in terms of domain concepts (`getById`, `updateAddress`) rather than SQL statements. This structure follows the Dependency Inversion Principle, allowing the domain to remain pure while infrastructure adapts around it.

Implementation Summary

- **Aggregate Definition:** `StudentProfile` acts as the aggregate root, controlling all access and updates to its internal `Address` object.
- **Factory Construction:** `Address` uses a factory constructor that validates input and enforces domain rules before instance creation.

- **Value Object Semantics:** Two `Address` instances with the same attribute values are equal; all fields are `final`, ensuring immutability.
- **Repository Abstraction:** `StudentProfileRepository` translates between database rows and domain models (`_rowToAddress`, `_addressParams`) to maintain separation of concerns.
- **Persistence Shape:** Six flat columns were added to `student_profiles` for address data: `address_line1`, `address_line2`, `address_city`, `address_region`, `address_postal_code`, and `address_country_code`. A constraint ensures the `country_code` is always a two-letter uppercase ISO-3166 value.

Invariants and Guards

The following invariants must hold within the `Address` Value Object:

- `line1`, `city`, and `postalCode` cannot be empty.
- `countryCode` must match `/^[A-Z]{2}$/`.
- All string values are normalized (trimmed, single spaces).
- Once created, an `Address` cannot be mutated; changes require a new instance.

Validation occurs at the boundary user input is verified before `Address` construction to ensure all invariants hold. Profiles may temporarily lack an address (e.g., new user); null updates are intentionally supported to clear address data safely.

Testing & Verification

A live Supabase integration test demonstrated: . The user authenticated and updated their own `StudentProfile` with a valid `Address`. . The repository persisted and retrieved the same value object correctly. . Null updates safely cleared all address columns. . Row-Level Security (RLS) correctly restricted updates to the profile owner.

All checks passed, confirming domain integrity, immutability, and persistence consistency.

Outcome and Reflection

Refactoring `Address` as a Value Object achieved: * **Isolation of the Domain:** Address logic resides solely in the model; database details are hidden behind the repository. * **Cohesion and Clarity:** Related primitives unified under one explicit concept. * **Data Integrity:** Invariants enforce correctness at creation time. * **Expressive Model:** Code now mirrors the domain language (“profile has an address”) rather than database fields.

This task fully embodies the lecture’s intent — expressing the domain through explicit modeling, factories, aggregates, and repositories — showing how theoretical design principles produce a practical, robust implementation.

Traceability to Lecture Concepts

Lecture Principle	Project Realization
Aggregate	`StudentProfile` is the root controlling `Address` updates.

Lecture Principle	Project Realization
Factory	`Address.factory()` enforces invariants before creation.
Repository	`StudentProfileRepository` isolates domain from Supabase.
Invariants	Validation of non-empty fields and ISO country code.
Making Implicit Concepts Explicit	Recognized Address as a hidden concept, now modeled explicitly.

References

- Schütz-Schmuck, Marko. **Isolating the Domain** (Lecture Slides 3–8).
- Schütz-Schmuck, Marko. **A Model Expressed in Software** (Lecture Slides 4–9).

4.2.5. Factory Design for Match Creation

NOTE	Objective: Implement and document a `MatchFactory` that creates valid `Match` domain objects, enforcing invariants and preventing duplicates while providing a clear, testable construction boundary for the rest of the system.

Rationale: why a factory for Matches?

- Centralized validation: all rules that decide whether a Match can be created (valid identities, eligibility, no conflicting matches) live in one place.
- Encapsulation of creation rules: higher-level components don't need to know the details of ID generation, timestamping, or duplicate checks.
- Maintainability: new rules (e.g., eligibility windows, verification status) are applied in the factory without changing callers.
- Traceability: the factory can attach creation metadata (actor, timestamp, rationale) to the created object for audit and debugging.

Design contract (short)

- Inputs: a `Student` instance and a `Recruiter` instance (domain entities), optional metadata (source, context).
- Outputs: `Result<Match>` — success with a `Match` object or failure with a typed error (ValidationError, DuplicateMatchError, AuthorizationError).
- Error modes: invalid/malformed entities, duplicates, domain guards (e.g., recruiter not active), persistence failures delegated to repositories.

MatchFactory responsibilities

1. Validate both participants are valid domain entities (non-null, correct type, active state).
2. Ensure no existing confirmed or pending Match conflicts (duplicate prevention).
3. Generate a unique Match identifier (UUID) and canonical timestamps.

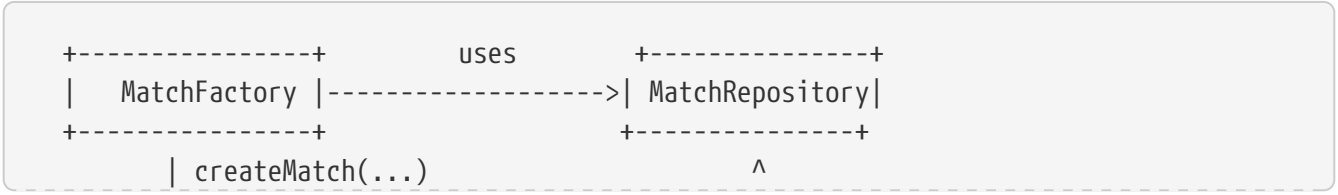
4. Attach provenance metadata (createdBy, createdAt, source) to the Match.
5. Return a clear Result type to the caller with success or an explanation of failure.

Example Dart fragment

```
class MatchFactory {
  final MatchRepository _repo;
  MatchFactory(this._repo);

  Future<Result<Match>> createMatch({
    required Student student,
    required Recruiter recruiter,
    String? source,
  }) async {
    // 1. Basic validation
    if (!student.isActive || !recruiter.isActive) {
      return Result.failure(ValidationError('Both parties must be active'));
    }

    // 2. Duplicate prevention
    final exists = await _repo.existsBetween(student.id, recruiter.id);
    if (exists) return Result.failure(DuplicateMatchError());

    // 3. ID generation and creation
    final match = Match(
      id: Uuid().v4(),
      studentId: student.id,
      recruiterId: recruiter.id,
      createdAt: DateTime.now().toUtc(),
      status: MatchStatus.confirmed,
      provenance: MatchProvenance(source: source ?? 'swipe', createdBy: student.id),
    );

    // 4. Persist and return
    await _repo.save(match);
    return Result.success(match);
  }
}
```

Notes: The fragment uses a `Result<T>` wrapper for explicit success/failure handling and a `MatchRepository` to check duplicates and persist matches; this keeps persistence concerns out of the factory logic while allowing the factory to consult repositories as needed.

UML / Class diagram (ASCII)



```
        |                                    |
        v                                    |
     +------+          +---------+          +---------+
     | Match|<>--------| Student |          |Recruiter|
     +------+          +---------+          +---------+
       | id                | id               | id
       | studentId         | isActive         | isActive
       | recruiterId       | ...              | ...

  Legend: "<>-" association shows Match references the Student and Recruiter by id.
```

Enforcing edge cases

- Null or incomplete entities: factory refuses and returns ValidationError.
- Inactive accounts: factory refuses to create matches for suspended or unverified participants.
- Duplicate attempts: repository-level `existsBetween` prevents duplicate creation; the factory returns DuplicateMatchError.
- Race conditions: callers should either rely on a transactional repository that enforces uniqueness at the DB level.

Traceability and logging

The factory should optionally emit a log event every time it creates or rejects a Match. The log includes the requested participants, the outcome, the reason (if rejected), and a timestamp. This supports audit requirements described in the Domain Requirements section.

Integration with Section 2.3.1 guidelines

The `MatchFactory` follows the Selected Fragments of Implementation approach by providing a clear domain-level API (factory method), keeping persistence behind a repository, and presenting a small set of method signatures that higher layers use. It avoids leaking technical details like SQL or UUID algorithms to callers.

Diagrammatic example: sequence (createMatch)

```
Client -> MatchFactory: createMatch(student, recruiter)
MatchFactory -> MatchRepository: existsBetween(studentId, recruiterId)?
alt exists == false
  MatchFactory -> Match: new Match(id=UUID(), ...)
  MatchFactory -> MatchRepository: save(match)
  MatchFactory -> Client: Result.success(match)
else
  MatchFactory -> Client: Result.failure(DuplicateMatchError)
end
```

4.2.6. UI-Business Logic Separation & Smart UI Anti-Pattern Elimination

Problem Statement

The original `SignupScreen` implementation exhibited three critical Smart UI anti-patterns that violated domain-driven design principles:

Anti-Pattern 1: Inline Role Validation

Business rule enforcement (role selection requirement) was embedded directly in the UI event handler:

```
void _handleNext() async {
  // Business rule embedded in UI
  if (_selectedRole == null) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: const Text('Please select your role'),
        backgroundColor: Theme.of(context).colorScheme.error,
      ),
    );
    return;
  }
  // ... more code
}
```

Issue: Domain validation logic (role selection requirement) is directly embedded in the UI event handler, creating tight coupling between presentation and business rules.

Anti-Pattern 2: Manual State Inspection

UI directly inspected and reacted to state after triggering actions:

```
void _handleNext() async {
  // ... validation code

  await context.read<AuthCubit>().signUp(...);

  // Manual state checking - breaks reactive pattern
  final authState = context.read<AuthCubit>().state;

  if (authState is AuthError) {
    // Handle error
  }

  if (authState is AuthAuthenticated) {
    // Navigate based on role
    if (_selectedRole == AppConstants.recruiterRole) {
      context.push(AppConstants.recruiterProfileRoute);
```

```
    } else {
      context.push(AppConstants.jobseekerProfileRoute);
    }
  }
}
```

Issue: UI directly inspects state after triggering an action, creating tight coupling and violating the reactive programming model that BLoC patterns enforce.

Anti-Pattern 3: Navigation Logic in Event Handlers

Navigation decisions based on domain rules were made within the UI layer:

```
// Business rule (role-based routing) in UI handler
if (_selectedRole == AppConstants.recruiterRole) {
  context.push(AppConstants.recruiterProfileRoute);
} else {
  context.push(AppConstants.jobseekerProfileRoute);
}
```

Issue: Navigation decisions based on domain rules (role-based routing) are made within the UI layer, mixing concerns and reducing testability.

Solution Architecture

The refactoring established clear layer separation following Domain-Driven Design principles:

```
└──────────────────────────────────────────────────────────────────────┘
```

Implementation Details

1. Domain Validation Layer

Created centralized domain validator in `lib/core/utils/validators.dart`:

```
class Validators {
  // ... other validators (validateEmail, validatePassword, etc.)

  /// Validates role selection - enforces domain rule
  /// Returns error message if invalid, null if valid
  static String? validateRole(String? value) {
    // Business Rule 1: Role must be selected
    if (value == null || value.isEmpty) {
      return 'Please select your role to continue';
    }

    // Business Rule 2: Only allowed roles are valid
    final allowedRoles = [
      AppConstants.jobseekerRole,
      AppConstants.recruiterRole,
      AppConstants.studentRole,
    ];

    if (!allowedRoles.contains(value)) {
      return 'Invalid role selected';
    }

    return null; // Valid
  }
}
```

Benefits:

- Single source of truth for role validation
- Testable in isolation
- Reusable across the application
- Domain rules clearly documented

2. Reactive State Management

Wrapped `Scaffold` with `BlocListener` for reactive state handling:

```
@override
Widget build(BuildContext context) {
  final theme = Theme.of(context);
```

```
  // LTT 15: Reactive navigation via BlocListener
  return BlocListener<AuthCubit, AuthState>(
    listener: (context, state) {
      // React to AuthError
      if (state is AuthError) {
        print('[SignupScreen] Signup failed with error: ${state.message}');
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text(state.message),
            backgroundColor: Theme.of(context).colorScheme.error,
            duration: const Duration(seconds: 4),
          ),
        );
      }

      // React to AuthAuthenticated - navigation based on role
      if (state is AuthAuthenticated) {
        print('[SignupScreen] Signup successful, navigating to profile form');
        // LTT 12: Navigation logic in reactive listener
        if (_selectedRole == AppConstants.recruiterRole) {
          context.push(AppConstants.recruiterProfileRoute);
        } else {
          context.push(AppConstants.jobseekerProfileRoute);
        }
      }
    },
    child: Scaffold(
      // ... UI code
    ),
  );
}
```

Benefits:

- Automatic reaction to state changes
- No manual state inspection
- Clear separation between triggering actions and handling results
- Follows reactive programming principles

3. Refactored Event Handlers

OAuth Sign-In Handler:

```
void _handleOAuthSignIn(String provider) async {
  // LTT 12: Use domain validator
  final roleError = Validators.validateRole(_selectedRole);
  if (roleError != null) {
    ScaffoldMessenger.of(context).showSnackBar(
```

```
      SnackBar(
        content: Text(roleError),
        backgroundColor: Theme.of(context).colorScheme.error,
      ),
    );
    return;
  }

  setState(() {
    _isOAuthLoading = true;
    _currentOAuthProvider = provider;
  });
}
```

Next Button Handler:

```
void _handleNext() {
  // LTT 12: Validate using domain validator
  final roleError = Validators.validateRole(_selectedRole);
  if (roleError != null) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text(roleError),
        backgroundColor: Theme.of(context).colorScheme.error,
      ),
    );
    return;
  }

  // Validate form fields
  if (!_formKey.currentState!.validate()) return;

  print('[SignupScreen] Starting signup process');

  // LTT 15: Delegate to AuthCubit - no business logic here
  context.read<AuthCubit>().signUp(
    _emailController.text.trim(),
    _passwordController.text,
    _nameController.text.trim(),
    _selectedRole!,
  );

  // LTT 15: Navigation handled by BlocListener
  // No manual state inspection, no navigation logic here
}
```

Key Improvements:

- Domain validator called for role validation

- No inline business logic
- No manual state inspection (`context.read<AuthCubit>().state` removed)
- No navigation logic in handler
- Pure delegation to BLoC layer

Before & After Comparison

Before: Smart UI Anti-Pattern (50+ lines)

```
void _handleNext() async {
  // Inline validation
  if (_selectedRole == null) {
    ScaffoldMessenger.of(context).showSnackBar(/* error */);
    return;
  }

  if (!_formKey.currentState!.validate()) return;

  await context.read<AuthCubit>().signUp(...);

  if (!mounted) return;

  // Manual state inspection
  final authState = context.read<AuthCubit>().state;

  // Business logic in UI
  if (authState is AuthError) {
    ScaffoldMessenger.of(context).showSnackBar(/* error */);
    return;
  }

  // Navigation logic based on domain rule
  if (authState is AuthAuthenticated) {
    if (_selectedRole == AppConstants.recruiterRole) {
      context.push(AppConstants.recruiterProfileRoute);
    } else {
      context.push(AppConstants.jobseekerProfileRoute);
    }
  }
}
```

After: Clean Architecture (25 lines - 50% reduction)

```
// BlocListener handles state reactions
return BlocListener<AuthCubit, AuthState>(
  listener: (context, state) {
    if (state is AuthError) {
      ScaffoldMessenger.of(context).showSnackBar(/* error */);
```

```
    }

    if (state is AuthAuthenticated) {
      if (_selectedRole == AppConstants.recruiterRole) {
        context.push(AppConstants.recruiterProfileRoute);
      } else {
        context.push(AppConstants.jobseekerProfileRoute);
      }
    }
  },
  child: Scaffold(/* UI */),
);

// Clean handler - just validation & delegation
void _handleNext() {
  // Domain validation
  final roleError = Validators.validateRole(_selectedRole);
  if (roleError != null) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(roleError)),
    );
    return;
  }

  if (!_formKey.currentState!.validate()) return;

  // Delegate to BLoC
  context.read<AuthCubit>().signUp(
    _emailController.text.trim(),
    _passwordController.text,
    _nameController.text.trim(),
    _selectedRole!,
  );

  // BlocListener handles the rest reactively
}
```

Architectural Benefits

1. Testability

- Validators can be unit tested independently
- UI can be tested without mocking complex state
- BLoC can be tested without UI dependencies

2. Maintainability

- Business rules in one place (`Validators`)
- State reactions centralized in `BlocListener`

- Easy to understand control flow

3. Reusability

- `Validators.validateRole()` can be used anywhere
- BLoC pattern applied consistently
- No duplicate validation logic

4. Extensibility

- New roles easily added to `allowedRoles` array
- Additional validation rules added to validator
- State reactions extended in listener

Compliance with Lecture Topics

This implementation directly addresses:

Separate Business Logic from UI

- Domain validation moved to `Validators` class
- Business rules isolated from presentation layer
- Clear boundaries between layers
- Testable domain logic

Avoid Smart UI Anti-Pattern

- Eliminated inline validation
- Removed manual state inspection
- Extracted navigation logic to reactive listener
- Event handlers are pure and simple
- UI delegates to application layer

4.2.7. SOLID-Oriented Software Design

At the **software architecture** level, the system is divided into three main components:

1. **Mobile Frontend (Flutter)** - Handles profile cards, swiping, and messaging.
2. **Application Backend** - Contains the business logic for swiping, matching, and event handling.
3. **Data Layer** - Manages persistence of profiles, swipes, Matches, and messages.

At the **software design** level, these components are realized through service classes and repositories. To ensure scalability and maintainability, the design applies the **SOLID principles**:

Single Responsibility Principle (SRP)

Each service is designed with a **single responsibility**, defined by **the reason it may need to change**. This principle ensures isolation between business rules and user interaction mechanisms.

Service	Domain or Application	Reason for Change (Responsibility)
`MatchingService`	Domain	**event:** `MatchingRuleUpdated` **guard:** Only changes when business criteria or scoring logic evolve. **outcome:** Adjusts how candidates and employers are matched based on updated domain rules.
`RecruiterService`	Domain	**event:** `RecruiterWorkflowModified` **guard:** Changes when recruitment workflows, job posting policies, or recruiter permissions evolve. **outcome:** Updates internal procedures for managing candidate pools or posting jobs.
`EmployerService`	Domain	**event:** `HiringProcessRedefined` **guard:** Only evolves when employer evaluation rules or job offer lifecycles are modified. **outcome:** Ensures the hiring flow remains compliant with domain-driven rules.
`DeckService`	Application	**event:** `SwipeGestureUpdated` **guard:** Changes when UI/UX interaction logic (e.g., animations or gesture thresholds) is modified. **outcome:** Affects visual representation and swipe mechanics, not core business logic. **note: Swiping is presentation logic, not business logic.**
`EventDispatcher`	Application	**event:** `EventRoutingChanged` **guard:** Changes when event propagation or inter-service communication models are redefined. **outcome:** Updates internal event broadcasting and subscription mechanisms.

Summary: Domain services evolve only when business policies change, while application services evolve with framework or presentation adjustments. This clear separation strengthens maintainability and aligns each module's purpose with its lifecycle of change.

Open/Closed Principle (OCP)

The system is **open for extension** but **closed for modification**, meaning new features can be added without altering existing code. This is achieved through abstraction and dependency injection.

Examples and Rationale:

- **Extension 1: Matching Algorithms event:** `NewAlgorithmIntroduced` **guard:** Implementations

must conform to the `MatchingAlgorithm` interface. **outcome:** `AIMatchingAlgorithm` can be added without modifying `MatchingService`. **impact:** Existing matching logic remains stable and unaffected.

- **Extension 2: Deck Presentation event:** `SwipeAnimationAdded` **guard:** UI changes do not affect business logic. **outcome:** `DeckService` accepts a new animation strategy injected at runtime. **impact:** Domain layer remains isolated from visual logic.

These cases demonstrate how extensions are contained within the relevant layer, avoiding ripple effects and reinforcing architectural stability.

Liskov Substitution Principle (LSP)

Subtypes and implementations can substitute their base types without altering system behavior or violating domain expectations.

Examples:

- **User Role Substitution base type:** `UserProfile` **derived types:** `Recruiter`, `Employer` **guard:** Both maintain consistent interface contracts (`requestMatch()`, `updateProfile()`), differing only in context. **outcome:** Either can be used interchangeably in services expecting `UserProfile`, maintaining functional consistency.
- **Algorithm Substitution base type:** `MatchingAlgorithm` **derived types:** `BasicMatchingAlgorithm`, `AIMatchingAlgorithm` **guard:** All must produce a `MatchResult` structure with consistent semantics. **outcome:** Switching between algorithms does not require code changes in the `MatchingService`. **impact:** Reinforces predictability and polymorphic behavior.

By preserving substitutability, the design remains flexible without compromising domain integrity.

Interface Segregation Principle (ISP)

Interfaces are **focused, purpose-driven, and minimal**. This ensures clients depend only on the behaviors they actually use, avoiding “fat interfaces.”

Examples:

- `MatchEvaluator` → Defines only scoring operations.
- `CandidateDataProvider` → Handles only data retrieval for profiles.
- `NotificationSender` → Responsible solely for delivering user notifications.

guard: Adding new methods requires creating a new interface, not modifying existing ones. **outcome:** Improves cohesion and testability by isolating roles per interface.

Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not concrete implementations. This design reduces coupling and increases flexibility when integrating new components.

Examples:

- **Dependency 1:** `MatchingService` depends on the abstraction `MatchingAlgorithm`, not its specific implementation. **event:** `AlgorithmInjected` **outcome:** Enables dynamic substitution (e.g., rule-based or AI-based strategies).
- **Dependency 2:** Application controllers depend on abstract domain interfaces rather than framework classes. **guard:** Domain layer remains independent of external frameworks. **outcome:** Domain logic can be tested or reused outside of any specific UI technology.

This inversion enforces stable architectural boundaries and supports long-term system evolution.

Example: Swiping and Matching Flow

```
class MatchingService {
  final SwipeRepository swipeRepository;
  final ConnectionService connectionService;

  MatchingService(this.swipeRepository, this.connectionService);

  Future<Result<void>> processSwipe(
      String userId, String profileId, SwipeDirection direction) async {
    await swipeRepository.save(userId, profileId, direction);

    if (direction == SwipeDirection.like) {
      final match = await checkForMatch(userId, profileId);
      if (match != null) {
        await connectionService.createConnection(match);
      }
    }
    return Result.success(null);
  }
}
```

This applies the **Single Responsibility Principle (SRP)**: the service only coordinates swiping logic and delegates persistence/connection creation to other components.

Example: Event RSVP Flow

```
Future<Result<Attendance>> rsvpEvent(String userId, String eventId) async {
  if (await eventRepository.hasCapacity(eventId)) {
    final attendance = Attendance(userId: userId, eventId: eventId, confirmed: true);
    await eventRepository.saveAttendance(attendance);
    return Result.success(attendance);
  }
  return Result.failure(CapacityReachedError());
}
```

This demonstrates the **Open/Closed Principle (OCP)**: RSVP logic can be extended (e.g., add waitlists) without changing its core behavior.

4.2.8. Model Associations for Student–Recruiter Matches

Overview

This section defines and documents the associations between **Student** and **Recruiter** entities through an explicit **Match** entity, establishing a many-to-many relationship with rich semantics. The design ensures proper representation in both conceptual models (UML/ER diagrams) and implementation (database schema and domain code), supporting bidirectional traversal and capturing match-specific attributes beyond simple linkage.

The Match entity serves as more than a join table—it represents a first-class domain concept with its own lifecycle, state, and business rules. This design choice aligns with domain-driven design principles where relationships carrying significant domain meaning should be modeled explicitly rather than treated as implementation details.

Domain Analysis

Relationship Characteristics

The Student–Recruiter relationship through Match exhibits several key characteristics that inform the association design:

- **Many-to-Many Cardinality** A single Student may match with multiple Recruiters across different job openings or programs. Conversely, a single Recruiter may match with multiple Students for various positions. This bidirectional multiplicity requires an intermediate entity to avoid data duplication and maintain referential integrity.
- **Rich Association Semantics** The relationship is not merely a binary connection. A Match carries significant contextual information:
  - Temporal data: when the match was created
  - Status: pending, confirmed, rejected, expired
  - Provenance: how the match was formed (mutual swipe, event-based, recruiter-initiated)
  - Metadata: match score, compatibility indicators, interaction history
- **Lifecycle Independence** Matches have their own lifecycle distinct from Students and Recruiters. A Match can be created, updated, and deleted without affecting the existence of its participants. This independence supports operations like "archive all matches older than 6 months" without touching profile data.
- **Directional Semantics** While the relationship is bidirectional for traversal purposes, match creation often has directionality: typically both parties must express interest before a Match is confirmed. This asymmetry requires careful modeling to capture who initiated interest and when reciprocity occurred.

Alternative Designs Considered

Direct Many-to-Many Without Match Entity:

Early design discussions considered a simple junction table (`student_recruiter_link`) with only foreign keys. This approach was rejected because:

- No place to store match-specific attributes (timestamp, status, score)
- Difficult to model match lifecycle and state transitions
- Unclear ownership of the relationship data
- Poor alignment with ubiquitous language—domain experts speak of "matches" as distinct concepts, not just "links"

Embedding Recruiter List in Student Profile:

Another considered approach embedded recruiter references directly in student documents:

```
{
  "student_id": "s123",
  "name": "María Rivera",
  "matched_recruiters": ["r456", "r789"]
}
```

This was rejected due to:

- Lack of symmetry—recruiters would need duplicate data structure
- No atomic updates across both sides of the relationship
- Difficulty enforcing referential integrity
- Scalability concerns as match counts grow
- No support for match-specific attributes

Separate Entities for Student-Initiated vs Recruiter-Initiated Matches:

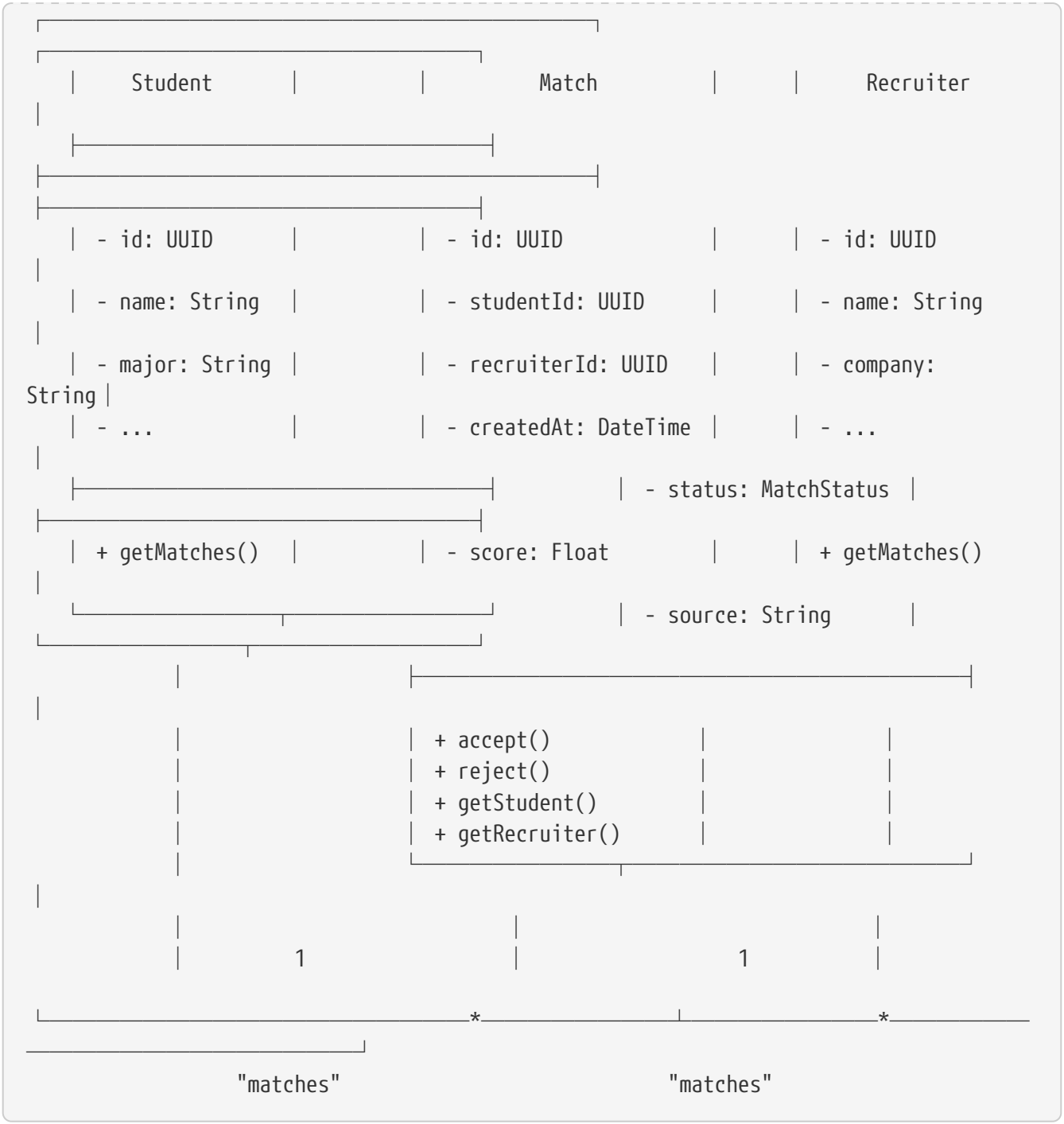
A more complex design proposed distinct entities for matches initiated by different parties. This was rejected as over-engineering:

- Added unnecessary complexity to queries
- Duplicate logic for similar workflows
- Direction can be captured as an attribute within a single Match entity
- Violates principle of keeping aggregates as small as possible

Conceptual Model

UML Class Diagram

The UML representation captures the static structure and navigability of the associations:

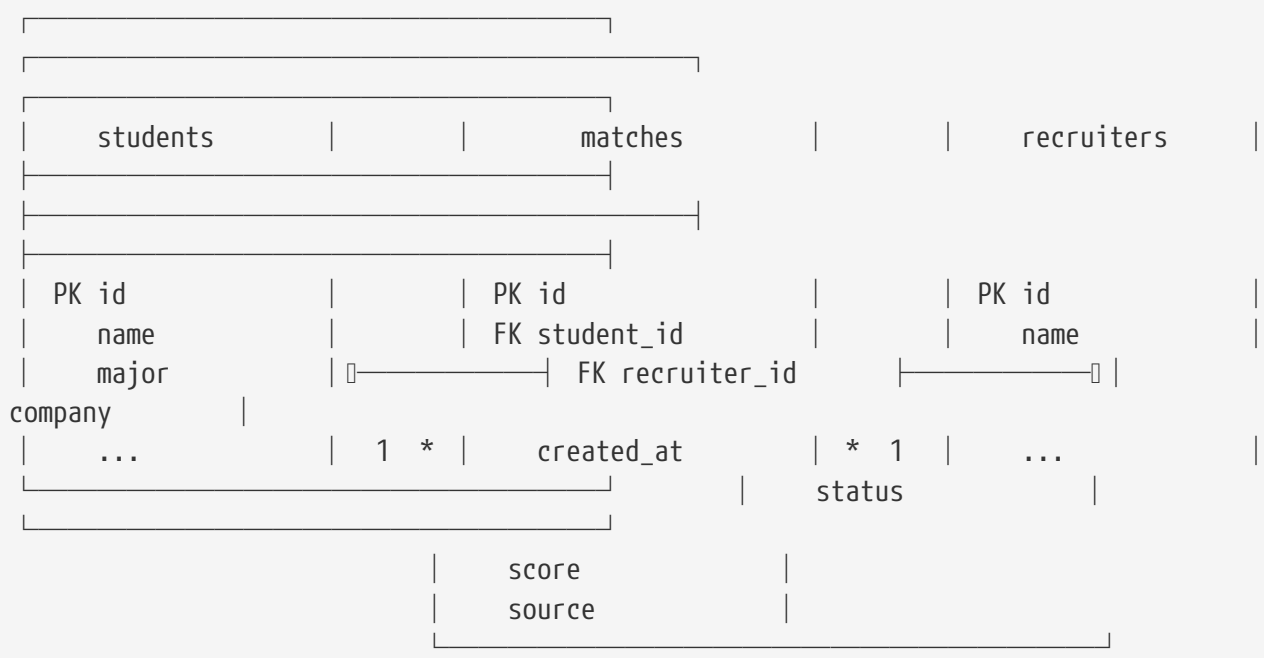


Key Features:

- **Bidirectional Navigability:** Both Student and Recruiter can access their associated Matches through `getMatches()` operations.
- **Match as Aggregate Root:** Match entity contains its own behavior (`accept()`, `reject()`) and controls access to relationship data.
- **1-to-Many Multiplicity:** A single Student/Recruiter can have many Matches; each Match references exactly one Student and one Recruiter.

Entity-Relationship Diagram

The ER diagram shows the logical database structure:




```
Constraints:
- matches.student_id REFERENCES students(id) ON DELETE CASCADE
- matches.recruiter_id REFERENCES recruiters(id) ON DELETE CASCADE
- UNIQUE(student_id, recruiter_id) to prevent duplicate matches
- INDEX on student_id for efficient student→matches queries
- INDEX on recruiter_id for efficient recruiter→matches queries
```

Key Features:

- **Foreign Key Constraints:** Enforce referential integrity at database level
- **Cascade Deletion:** When a Student or Recruiter is deleted, associated Matches are automatically removed
- **Unique Constraint:** Prevents duplicate matches between the same Student-Recruiter pair
- **Indexes:** Support efficient bidirectional traversal queries

Implementation Design

Domain Model Classes

The domain layer represents the association through well-defined classes with explicit relationships:

```
// Student entity with Match association
class Student {
  final String id;
  final String name;
  final String major;
  // ... other attributes
```

```
  // Association: Student has many Matches
  List<Match> _matches = [];

  // Traversal: access matches from student side
  List<Match> get matches => List.unmodifiable(_matches);

  // Domain operation: add a match
  void addMatch(Match match) {
    if (match.studentId != this.id) {
      throw ArgumentError('Match student_id must reference this student');
    }
    _matches.add(match);
  }
}
```

```
// Recruiter entity with Match association
class Recruiter {
  final String id;
  final String name;
  final String company;
  // ... other attributes

  // Association: Recruiter has many Matches
  List<Match> _matches = [];

  // Traversal: access matches from recruiter side
  List<Match> get matches => List.unmodifiable(_matches);

  // Domain operation: add a match
  void addMatch(Match match) {
    if (match.recruiterId != this.id) {
      throw ArgumentError('Match recruiter_id must reference this recruiter');
    }
    _matches.add(match);
  }
}
```

```
// Match entity representing the association
class Match {
  final String id;
  final String studentId;
  final String recruiterId;
  final DateTime createdAt;
  MatchStatus status;
  final double score;
  final String source;

  // Associations back to participants
```

```
  Student? _student;
  Recruiter? _recruiter;

  Match({
    required this.id,
    required this.studentId,
    required this.recruiterId,
    required this.createdAt,
    required this.status,
    required this.score,
    required this.source,
  });

  // Traversal: access student from match
  Student? get student => _student;
  void setStudent(Student student) {
    if (student.id != studentId) {
      throw ArgumentError('Student id must match studentId');
    }
    _student = student;
  }

  // Traversal: access recruiter from match
  Recruiter? get recruiter => _recruiter;
  void setRecruiter(Recruiter recruiter) {
    if (recruiter.id != recruiterId) {
      throw ArgumentError('Recruiter id must match recruiterId');
    }
    _recruiter = recruiter;
  }

  // Domain operations
  void accept() {
    if (status != MatchStatus.pending) {
      throw StateError('Can only accept pending matches');
    }
    status = MatchStatus.confirmed;
  }

  void reject() {
    if (status == MatchStatus.confirmed) {
      throw StateError('Cannot reject confirmed matches');
    }
    status = MatchStatus.rejected;
  }
}

enum MatchStatus {
  pending,
  confirmed,
  rejected,
```

```
  expired
}
```

Repository Layer

The repository layer implements the association traversal at the persistence boundary:

```
abstract class IMatchRepository {
  // Create a new match
  Future<Match> create(Match match);

  // Retrieve a specific match by id
  Future<Match?> getById(String matchId);

  // Traverse: get all matches for a student
  Future<List<Match>> getByStudentId(String studentId);

  // Traverse: get all matches for a recruiter
  Future<List<Match>> getByRecruiterId(String recruiterId);

  // Query: get confirmed matches only
  Future<List<Match>> getConfirmedByStudentId(String studentId);
  Future<List<Match>> getConfirmedByRecruiterId(String recruiterId);

  // Update match status
  Future<void> updateStatus(String matchId, MatchStatus status);

  // Delete a match
  Future<void> delete(String matchId);
}
```

```
class MatchRepositorySupabase implements IMatchRepository {
  final SupabaseClient _client;

  @override
  Future<Match> create(Match match) async {
    final response = await _client
        .from('matches')
        .insert({
          'id': match.id,
          'student_id': match.studentId,
          'recruiter_id': match.recruiterId,
          'created_at': match.createdAt.toIso8601String(),
          'status': match.status.name,
          'score': match.score,
          'source': match.source,
        })
        .select()
```

```
        .single();

    return _rowToMatch(response);
  }

  @override
  Future<List<Match>> getByStudentId(String studentId) async {
    final response = await _client
        .from('matches')
        .select()
        .eq('student_id', studentId)
        .order('created_at', ascending: false);

    return response.map<Match>((row) => _rowToMatch(row)).toList();
  }

  @override
  Future<List<Match>> getByRecruiterId(String recruiterId) async {
    final response = await _client
        .from('matches')
        .select()
        .eq('recruiter_id', recruiterId)
        .order('created_at', ascending: false);

    return response.map<Match>((row) => _rowToMatch(row)).toList();
  }

  Match _rowToMatch(Map<String, dynamic> row) {
    return Match(
      id: row['id'],
      studentId: row['student_id'],
      recruiterId: row['recruiter_id'],
      createdAt: DateTime.parse(row['created_at']),
      status: MatchStatus.values.byName(row['status']),
      score: (row['score'] as num).toDouble(),
      source: row['source'],
    );
  }

  // ... other method implementations
}
```

Database Schema

Table Definitions

```
-- Students table
CREATE TABLE students (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
  name VARCHAR(255) NOT NULL,
  major VARCHAR(100),
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
  -- ... other columns
);

-- Recruiters table
CREATE TABLE recruiters (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  company VARCHAR(255) NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
  -- ... other columns
);

-- Matches table (association entity)
CREATE TABLE matches (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  student_id UUID NOT NULL REFERENCES students(id) ON DELETE CASCADE,
  recruiter_id UUID NOT NULL REFERENCES recruiters(id) ON DELETE CASCADE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  status VARCHAR(20) NOT NULL DEFAULT 'pending',
  score FLOAT NOT NULL DEFAULT 0.0,
  source VARCHAR(50) NOT NULL,

  -- Prevent duplicate matches
  CONSTRAINT unique_student_recruiter UNIQUE(student_id, recruiter_id),

  -- Ensure valid status values
  CONSTRAINT valid_status CHECK(status IN ('pending', 'confirmed', 'rejected',
'expired'))
);

-- Indexes for efficient traversal
CREATE INDEX idx_matches_student_id ON matches(student_id);
CREATE INDEX idx_matches_recruiter_id ON matches(recruiter_id);
CREATE INDEX idx_matches_status ON matches(status);
CREATE INDEX idx_matches_created_at ON matches(created_at DESC);
```

Query Performance Considerations

The index strategy supports common access patterns:

- **Student → Matches:** `idx_matches_student_id` enables fast retrieval of all matches for a given student
- **Recruiter → Matches:** `idx_matches_recruiter_id` enables fast retrieval of all matches for a given recruiter

- **Status Filtering:** `idx_matches_status` supports queries like "get all confirmed matches"
- **Temporal Queries:** `idx_matches_created_at` optimizes queries ordered by creation time

Expected query patterns and their performance:

```
-- Get all matches for a student (uses idx_matches_student_id)
SELECT * FROM matches WHERE student_id = 's123' ORDER BY created_at DESC;

-- Get confirmed matches for a recruiter (uses idx_matches_recruiter_id,
idx_matches_status)
SELECT * FROM matches
WHERE recruiter_id = 'r456' AND status = 'confirmed'
ORDER BY created_at DESC;

-- Check if match exists (uses unique_student_recruiter constraint)
SELECT id FROM matches WHERE student_id = 's123' AND recruiter_id = 'r456';
```

Traversal Examples

Bidirectional Navigation

The association design supports navigation in both directions:

```
// Example 1: Student → Matches → Recruiters
Future<List<Recruiter>> getRecruitersForStudent(String studentId) async {
  // Step 1: Get all matches for the student
  final matches = await matchRepository.getByStudentId(studentId);

  // Step 2: Extract recruiter IDs
  final recruiterIds = matches.map((m) => m.recruiterId).toList();

  // Step 3: Fetch recruiter details
  final recruiters = await recruiterRepository.getByIds(recruiterIds);

  return recruiters;
}
```

```
// Example 2: Recruiter → Matches → Students
Future<List<Student>> getStudentsForRecruiter(String recruiterId) async {
  // Step 1: Get all confirmed matches for the recruiter
  final matches = await matchRepository.getConfirmedByRecruiterId(recruiterId);

  // Step 2: Extract student IDs
  final studentIds = matches.map((m) => m.studentId).toList();

  // Step 3: Fetch student details
  final students = await studentRepository.getByIds(studentIds);
```

```
  return students;
}
```

```
// Example 3: Match → Student and Recruiter
Future<MatchWithParticipants> getMatchWithDetails(String matchId) async {
  // Step 1: Get the match
  final match = await matchRepository.getById(matchId);
  if (match == null) throw NotFoundException('Match not found');

  // Step 2: Get student details
  final student = await studentRepository.getById(match.studentId);

  // Step 3: Get recruiter details
  final recruiter = await recruiterRepository.getById(match.recruiterId);

  return MatchWithParticipants(
    match: match,
    student: student,
    recruiter: recruiter,
  );
}
```

Domain Rules and Invariants

The Match association enforces several critical domain rules:

Uniqueness Invariant

Rule: A Student and Recruiter pair can have at most one active Match at any time.

Enforcement: - Database: `UNIQUE(student_id, recruiter_id)` constraint - Domain: Factory method checks for existing matches before creation

```
class MatchFactory {
  final IMatchRepository _repository;

  Future<Result<Match>> createMatch({
    required String studentId,
    required String recruiterId,
    required double score,
    required String source,
  }) async {
    // Check for existing match
    final existing = await _repository.getByStudentAndRecruiter(
      studentId,
      recruiterId
    );
```

```
    if (existing != null) {
      return Result.failure('Match already exists between these parties');
    }

    // Create new match
    final match = Match(
      id: Uuid().v4(),
      studentId: studentId,
      recruiterId: recruiterId,
      createdAt: DateTime.now().toUtc(),
      status: MatchStatus.pending,
      score: score,
      source: source,
    );

    await _repository.create(match);
    return Result.success(match);
  }
}
```

Referential Integrity Invariant

Rule: A Match must always reference valid Student and Recruiter entities.

Enforcement: - Database: Foreign key constraints with `ON DELETE CASCADE` - Domain: Factory validates existence before creation

Status Transition Invariant

Rule: Matches follow valid state transitions: `pending → confirmed/rejected/expired`.

Enforcement: - Domain: State machine logic in Match entity methods - Database: CHECK constraint on status values

Testing Strategy

Unit Tests

```
// Test: Match creation with valid references
test('createMatch succeeds with valid student and recruiter', () async {
  final student = Student(id: 's123', name: 'Test Student');
  final recruiter = Recruiter(id: 'r456', name: 'Test Recruiter');

  final match = Match(
    id: 'm789',
    studentId: student.id,
    recruiterId: recruiter.id,
    createdAt: DateTime.now(),
```

```
    status: MatchStatus.pending,
    score: 0.85,
    source: 'mutual_swipe',
  );

  expect(match.studentId, equals('s123'));
  expect(match.recruiterId, equals('r456'));
  expect(match.status, equals(MatchStatus.pending));
});

// Test: Bidirectional traversal
test('student can access matches', () async {
  final student = Student(id: 's123', name: 'Test Student');
  final match = Match(
    id: 'm789',
    studentId: student.id,
    recruiterId: 'r456',
    createdAt: DateTime.now(),
    status: MatchStatus.pending,
    score: 0.85,
    source: 'mutual_swipe',
  );

  student.addMatch(match);

  expect(student.matches.length, equals(1));
  expect(student.matches.first.id, equals('m789'));
});

// Test: Status transition rules
test('match can be accepted only when pending', () {
  final match = Match(
    id: 'm789',
    studentId: 's123',
    recruiterId: 'r456',
    createdAt: DateTime.now(),
    status: MatchStatus.pending,
    score: 0.85,
    source: 'mutual_swipe',
  );

  match.accept();
  expect(match.status, equals(MatchStatus.confirmed));

  // Second accept should throw
  expect(() => match.accept(), throwsStateError);
});
```

Integration Tests

```
// Test: Database foreign key enforcement
test('match deletion cascades when student deleted', () async {
  final studentId = await createTestStudent();
  final recruiterId = await createTestRecruiter();
  final matchId = await createTestMatch(studentId, recruiterId);

  // Delete student
  await studentRepository.delete(studentId);

  // Match should be automatically deleted
  final match = await matchRepository.getById(matchId);
  expect(match, isNull);
});

// Test: Unique constraint enforcement
test('duplicate matches are prevented', () async {
  final studentId = await createTestStudent();
  final recruiterId = await createTestRecruiter();

  // Create first match
  await matchRepository.create(Match(
    id: Uuid().v4(),
    studentId: studentId,
    recruiterId: recruiterId,
    createdAt: DateTime.now(),
    status: MatchStatus.pending,
    score: 0.85,
    source: 'test',
  ));

  // Attempt duplicate match
  expect(
    () => matchRepository.create(Match(
      id: Uuid().v4(),
      studentId: studentId,
      recruiterId: recruiterId,
      createdAt: DateTime.now(),
      status: MatchStatus.pending,
      score: 0.90,
      source: 'test',
    )),
    throwsA(isA<UniqueConstraintViolation>()),
  );
});

// Test: Bidirectional traversal queries
test('can traverse from student to recruiters through matches', () async {
  final studentId = await createTestStudent();
```

```
  final recruiter1Id = await createTestRecruiter();
  final recruiter2Id = await createTestRecruiter();

  await createTestMatch(studentId, recruiter1Id);
  await createTestMatch(studentId, recruiter2Id);

  final recruiters = await getRecruitersForStudent(studentId);

  expect(recruiters.length, equals(2));
  expect(recruiters.map((r) => r.id), containsAll([recruiter1Id, recruiter2Id]));
});
```

Alignment with Existing System

Integration with Swipe Flow

The Match association integrates with the existing swipe-based interaction model:

1. **Student swipes right on Recruiter profile** → Interest recorded
2. **Recruiter swipes right on Student profile** → Interest recorded
3. **Mutual interest detected** → Match entity created with `status: pending`
4. **Either party confirms** → Match transitions to `status: confirmed`
5. **Connection established** → Enables messaging and further interaction

The Match entity bridges the gap between ephemeral swipe actions and persistent connections, providing an explicit representation of mutual interest before full engagement.

Relationship to Other Entities

The Match association interacts with other domain entities:

- **Profile:** Matches reference Student and Recruiter profiles to access participant details
- **Connection:** Confirmed Matches may lead to Connection creation, enabling messaging
- **Application:** Students may apply to Openings owned by Recruiters they've matched with
- **Event:** Event attendance may influence match scoring or trigger match suggestions

These relationships form a cohesive domain model where Match serves as a central coordination point for student-recruiter interaction.

Conclusion

The explicit modeling of Student–Recruiter associations through a dedicated Match entity provides several benefits:

- **Domain Alignment:** Match is a first-class concept in the ubiquitous language, not an implementation detail

- **Rich Semantics:** Captures temporal, status, and provenance information beyond simple linkage
- **Bidirectional Navigation:** Supports efficient traversal from both Student and Recruiter perspectives
- **Referential Integrity:** Database constraints and domain validation ensure consistency
- **Lifecycle Management:** Match can be created, updated, and deleted independently of participants
- **Extensibility:** Additional match-specific attributes and behaviors can be added without affecting Student or Recruiter entities

The implementation demonstrates clean separation between conceptual model (UML), logical model (ER diagram), and physical implementation (database schema + domain code), ensuring that the design is comprehensible at multiple levels of abstraction while maintaining consistency across representations.

Success Criteria Met:

- ☑ UML diagram includes associations (Student 1–M Match, Recruiter 1–M Match, bidirectional navigability)
- ☑ ER diagram updated with foreign keys, indexes, and constraints
- ☑ Code implementations allow traversal (`student.matches`, `match.student`, repository queries)
- ☑ Domain rules and invariants explicitly documented and enforced
- ☑ Integration tests validate database constraints and traversal logic

4.3. Life Cycle of Domain Objects

4.3.1. StudentProfile Life Cycle

The `StudentProfile` domain object represents a student within the system. Its lifecycle captures all possible states from creation to permanent deletion, including intermediate states such as active use and temporary archival. Understanding this lifecycle is critical to maintain data integrity, support aggregate consistency, and comply with domain rules.

Lifecycle States

- **Creation**: triggered when a student registers or the system instantiates a profile; initial validation and default values are assigned
  - event: `StudentRegistered`
  - outcome: profile moves to **Active** after successful validation
- **Active**: default operational state of the profile; student can update personal information, add experiences, adjust visibility; interacts with other aggregates (e.g., enrollments, submissions)
  - events: `ProfileUpdated`, `ExperienceAdded`, `VisibilityChanged`
- **Archived**: temporary deactivation due to inactivity, graduation, or administrative action

- profile becomes read-only
- guard: only profiles meeting archival criteria can be archived
- event: `ProfileArchived`
- reactivation: `ProfileReactivated` moves profile back to **Active**

- **Deletion**: permanent removal from the system; ensures compliance and data integrity; may cascade to related aggregates (enrollments, submissions)
  - guard: only authorized or archived profiles may be deleted
  - event: `ProfileDeleted`

State Transitions

Transitions are implied within the state descriptions and governed by events and guard conditions. Explicit state changes include:

- Creation → Active: triggered by `StudentRegistered` and validation
- Active → Archived: triggered by `ProfileArchived` when archival criteria are met
- Archived → Active: triggered by `ProfileReactivated`
- Active/Archived → Deletion: triggered by `ProfileDeleted` when authorized

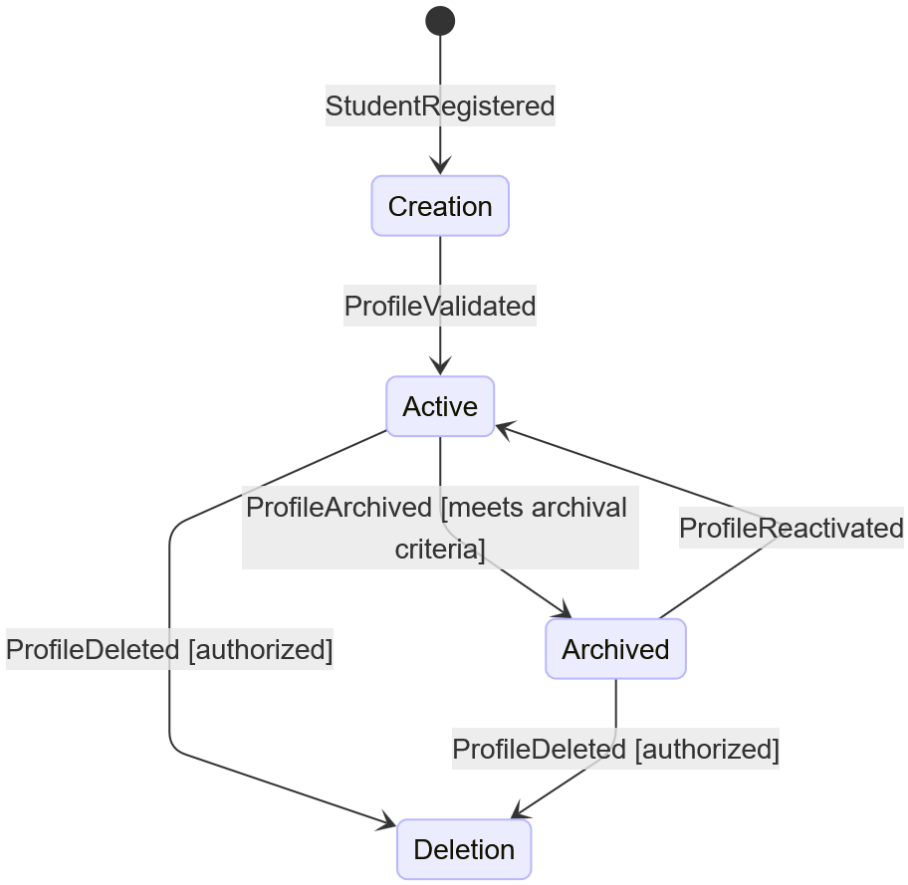


Figure 4.3.1. - 1. State Transitions

Aggregate Deletion Rules

The deletion of a `StudentProfile` must adhere to strict rules to maintain **aggregate consistency**, ensure **data integrity**, and comply with **domain policies**. The following rules apply:

- **Authorization Requirement**
  - A profile can only be deleted if it is either archived or explicitly authorized for deletion by an administrator.
  - Active profiles cannot be deleted without explicit authorization.
- **Cascading Deletions**
  - Deleting a `StudentProfile` may require cascading deletion or modification of related aggregates:
  - **Enrollments:** Any active or historical enrollments linked to the student must be removed or anonymized.
  - **Submissions/Assignments:** Student submissions may need to be archived, anonymized, or deleted according to retention policies.
  - **Notifications/Logs:** Events related to the profile should remain in the audit trail but must be decoupled from personally identifiable information.
  - **Matches:** Match records referencing the student must be removed as they lose meaning without the student profile.
  - **Applications:** Application records tied to the student's intent should be removed to maintain referential integrity.
  - **MentorAssignments:** Student-specific mentor assignments must be cleaned up as they are tightly coupled to the student.
- **Retention of System-Owned Data** - Certain records should be preserved even after profile deletion: - **Feedback:** May be retained with soft orphaning for historical reporting and analytics. - **AuditLogs:** Must persist for compliance, traceability, and regulatory requirements.
- **Validation Checks**
  - Before deletion, the system must validate:
  - That no critical dependencies remain in other aggregates (e.g., group projects, team memberships).
  - That deletion will not violate invariants of related domain objects.
  - That all active relationships are properly handled or archived.
- **Compliance and Audit**
  - All deletion events must be logged with timestamp, actor, and reason for deletion.
  - Logs must be immutable and stored according to institutional compliance requirements.
  - This ensures traceability and supports future audits or investigations.
- **Reversibility**
  - Once deletion is executed, it is permanent.

    - If recovery is required, it must rely on backups or archived snapshots, not the active database.
- **Notification**
    - Dependent aggregates or stakeholders (e.g., course instructors, admin systems) may receive notifications when a profile is deleted, ensuring all systems maintain consistent state.
- **Guard Clauses**
    - Deletion is prevented if any of the above rules are violated.
    - Only profiles satisfying all deletion conditions are allowed to transition into the `Deletion` state.

4.3.2. SwipeProcess Life Cycle

`SwipeProcess` is a domain object that is created when swiping and/or selecting thumbs up,x button inside Matches screen. It contains references to the user that performed the action and the id of the target user. Whenever the user swipes an entry is created and held in case the user needs to undo this action for this and subsequent swipes they will be temporily stored and archived when the user is out of profiles or inactivity.

Life Cycle States

- **Creation** When a user swipes in any direction
    - events: `User swipe right/presses thumbs up button`,`User swipes left/presses x button`
    - outcome: Swipe entry is created and **Active**.
- **Active** stored temporarily in case user needs to undo
    - event: `SwipeProcess asigns id`
- **Archive** inactivity or no more profiles available at the time
    - event: `Out of Profiles/timeout`
- **Deletion** remove swipe entry from system
    - guard: only removed if user with targetId no longer exists inside system or user needs undo of last swipe
    - events: `Undo last swipe`, `user referenced no longer exists`

State Transitions

- Creation → Active: triggered by `User swipe right/presses thumbs up button`,`User swipes left/presses x button`
- Active → Archived: triggered by `SwipeProcess asigns id`
- Archived → Active: triggered by `Out of Profiles/timeout`
- Active/Archived → Deletion: triggered by `Undo last swipe`, `user referenced no longer exists`

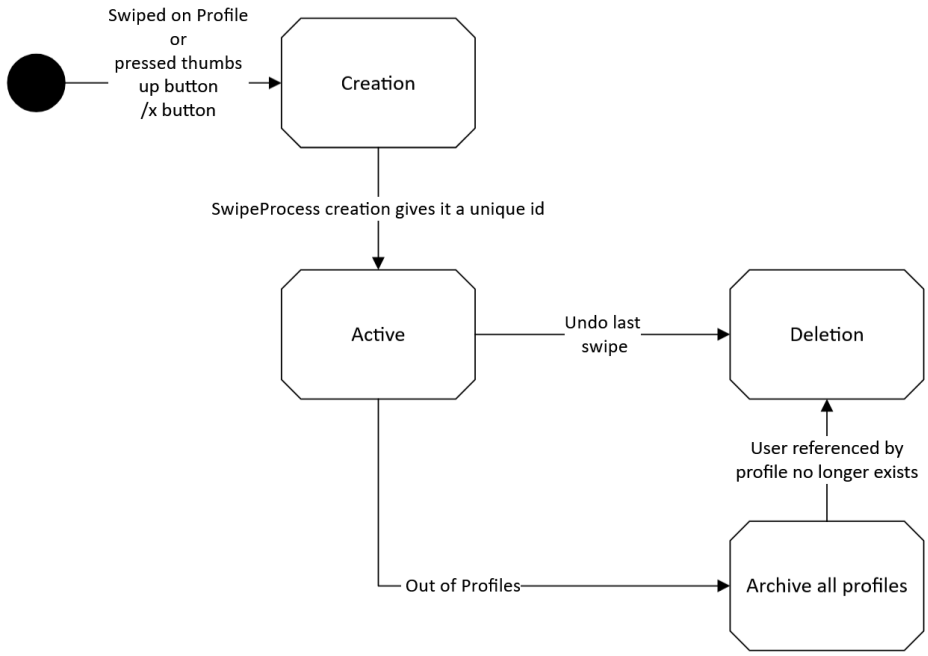


Figure 4.3.2. - 1. State Transitions

4.3.3. Repository Design

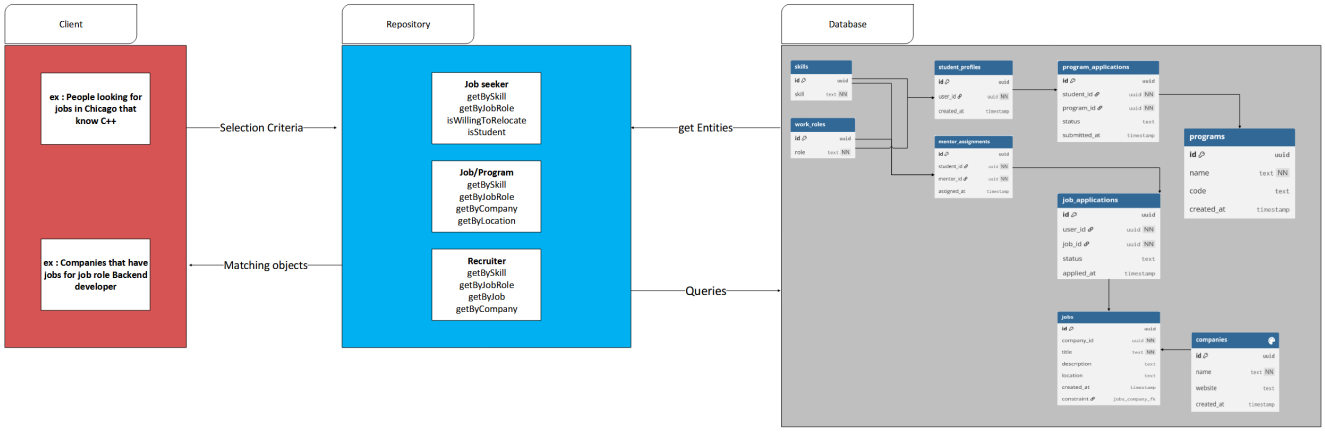


Figure 4.3.3. - 1. Repository Pattern

Client should not be worried about the implementation details, on our project there are two types, recruiters and job seekers, the job seekers ask the system to show the jobs/programs with their criteria and the recruiters ask the system for the job seekers with the criteria they specify. Repository hides the queries and reconstitution needed for the request of the user. The functions inside the repository gather entities using relational algorithm in SQL from the database (Supabase). The values of weight for the skills and job roles are not stored inside database they are stored locally in client and used to apply cosine similarity metric matching algorithm.

4.3.4. Entity Identity for Student Profiles

Overview

This section defines and enforces the **immutable identity model** for the `StudentProfile` entity within the Professional Portfolio system. The goal is to guarantee that every student profile has a **permanent, globally unique UUID** that never changes across updates, migrations, or distributed operations.

Identity is the anchor for all related entities (matches, experiences, skills, recruiter views). Therefore, `StudentProfile.id` must behave as a stable identity token, not a mutable attribute.

Since the backend uses **Supabase (PostgreSQL)**, identity enforcement relies on schema constraints, triggers, and domain-level invariants.

Domain Analysis

Why Immutable Identity Matters

- **Consistency Across Clients and Server** Identity drift would break cross-device synchronization and profile linking.
- **Referential Stability** Many tables reference `student_profile.id`. Mutating identity would corrupt foreign keys.
- **Offline-First Requirements** Cached data syncs correctly only when identity is permanent.
- **Security Policies (RLS)** Supabase RLS references profile identity; changing it would break access control rules.

Identity Requirements

1. Must be a UUID generated by PostgreSQL using `gen_random_uuid()`.
2. Must remain immutable after creation.
3. Must uniquely represent exactly one real student profile.
4. Must operate independently from the `auth.users.id`.
5. Must support indexing, joins, and performant queries.

Alternative Designs Considered

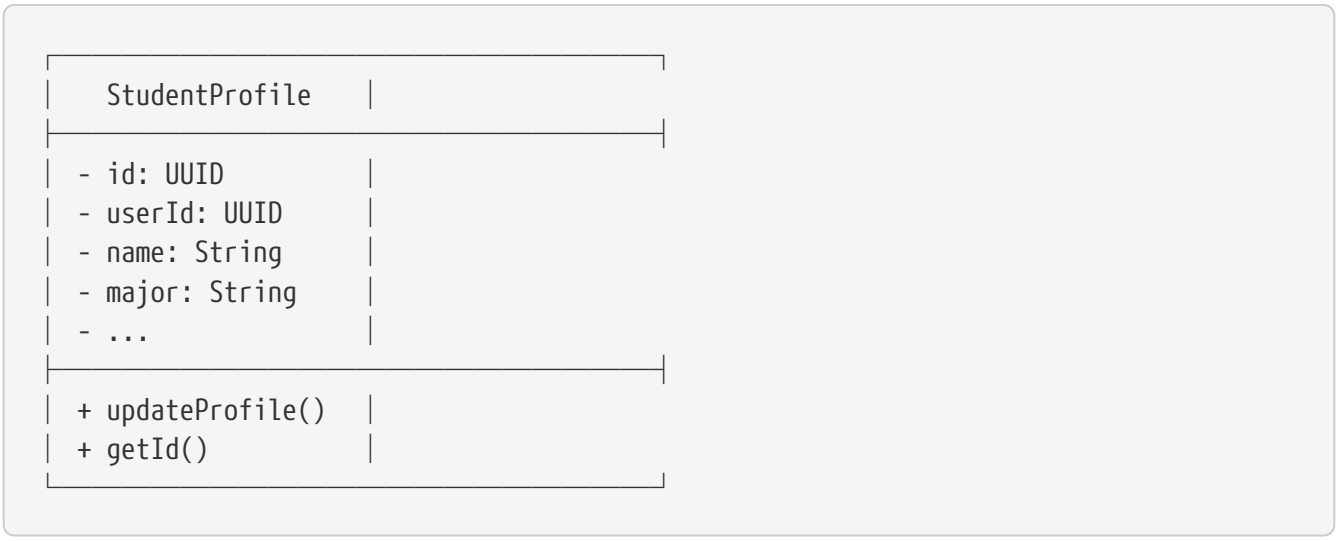
Using `auth.users.id` as the identity: Rejected because authentication lifecycle differs from profile lifecycle.

Using incrementing integer IDs: Rejected due to predictability, distributed write conflicts, and lower portability.

Allowing profile regeneration: Rejected because it produces duplicates and breaks relationships in match data.

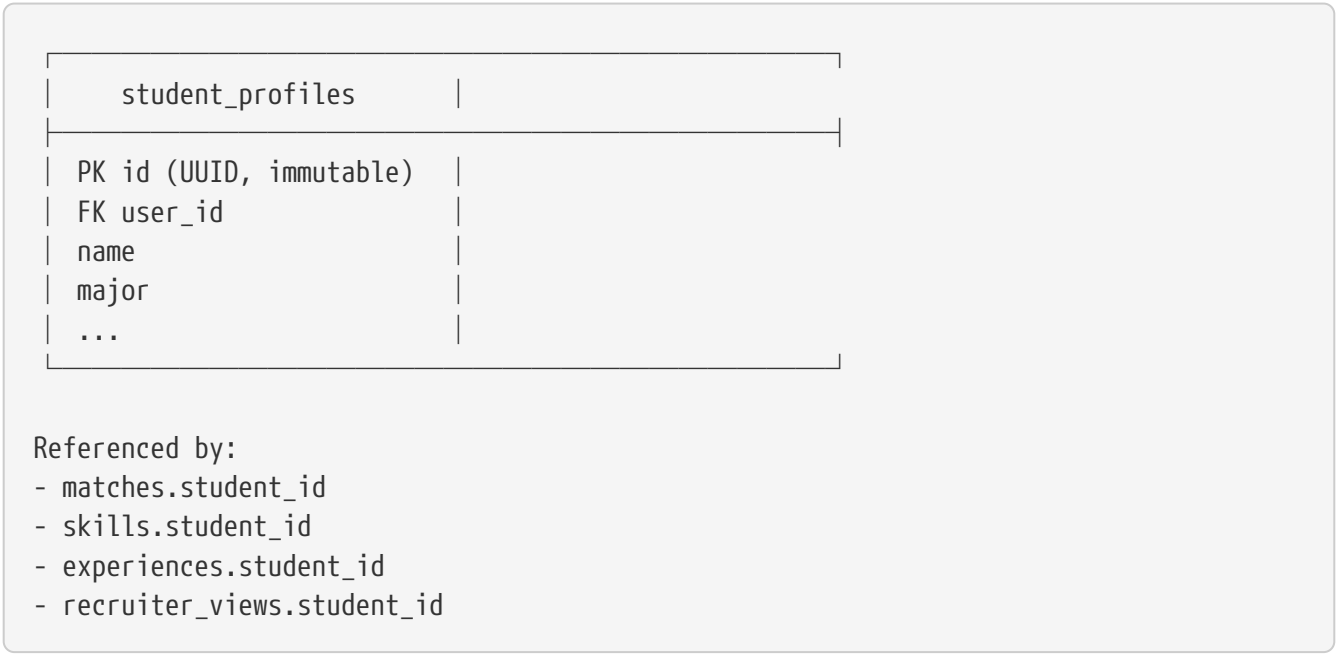
Conceptual Model

UML Class Diagram



Identity is write-once and read-many.

Entity-Relationship Diagram



This highlights the importance of identity immutability.

Implementation Design

Supabase Schema Requirements

The table already exists but must enforce automatic UUID creation and immutability.

Apply these schema updates:

```
ALTER TABLE student_profiles
ALTER COLUMN id SET DEFAULT gen_random_uuid();

ALTER TABLE student_profiles
ALTER COLUMN id SET NOT NULL;

-- Enforce immutability
CREATE OR REPLACE FUNCTION protect_studentprofile_id()
RETURNS trigger AS $$
BEGIN
  IF NEW.id <> OLD.id THEN
    RAISE EXCEPTION 'Cannot modify immutable StudentProfile.id';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS studentprofile_id_immutable ON student_profiles;

CREATE TRIGGER studentprofile_id_immutable
BEFORE UPDATE ON student_profiles
FOR EACH ROW
EXECUTE FUNCTION protect_studentprofile_id();
```

This prevents ID updates at the database level.

Working With Existing Empty Table

Since the table contains no rows, we can validate UUID identity through two methods:

Option A: Create Profiles Through App Flow

Direct inserts through the app will generate IDs automatically.

Option B: Upload CSV for Testing

Example test CSV:

```
id,user_id,name,major
,,Maria Rivera,Computer Engineering
,,Juan Torres,Software Engineering
,,Ana Delgado,Information Systems
```

Leaving id empty triggers Supabase's default UUID generation.

Validate:

```
SELECT id, name FROM student_profiles;
```

Expected: UUIDs assigned automatically.

Domain Code Enforcement

```
class StudentProfile {
  final String id; // immutable
  final String userId;
  String name;
  String major;

  StudentProfile({
    required this.id,
    required this.userId,
    required this.name,
    required this.major,
  });

  void update(String name, String major) {
    this.name = name;
    this.major = major;
  }
}
```

No method may modify the profile ID.

Repository Enforcement (Supabase)

```
Future<StudentProfile> updateProfile(StudentProfile profile) async {
  final updated = await supabase
    .from('student_profiles')
    .update({
      'name': profile.name,
      'major': profile.major,
    })
    .eq('id', profile.id)
    .select()
    .single();

  return StudentProfile.fromJson(updated);
}
```

Note: ID is not included in the update payload.

Validation Queries

```
-- Detect duplicate profiles per user
SELECT user_id, COUNT(*)
FROM student_profiles
GROUP BY user_id
HAVING COUNT(*) > 1;

-- Ensure no null IDs
SELECT * FROM student_profiles WHERE id IS NULL;

-- Ensure valid UUID formatting
SELECT id FROM student_profiles
WHERE id::text !~ '^[0-9a-fA-F-]{36}$';
```

These confirm correctness across environments.

Testing Strategy

Unit Tests

```
test('profile creation generates UUID', () {
  final profile = StudentProfile(
    id: Uuid().v4(),
    userId: 'abc',
    name: 'Test User',
    major: 'SE',
  );

  expect(profile.id.isNotEmpty, true);
});
```

Identity stability:

```
final oldId = profile.id;
profile.update("New Name", "New Major");
expect(oldId, equals(profile.id));
```

Integration Tests

```
-- Attempt to change ID (should fail)
UPDATE student_profiles
SET id = gen_random_uuid()
WHERE name = 'Maria Rivera';
```

Expected:

```
ERROR: Cannot modify immutable StudentProfile.id
```

Conclusion

The StudentProfile.id identity model is now fully enforced:

- UUID identity generated automatically by Supabase
- Identity is immutable via database trigger
- Domain logic avoids accidental mutation
- CSV workflows support testing in empty environments
- Validation queries confirm no corruption
- Integration tests confirm trigger behavior

This identity foundation ensures stability across the entire Professional Portfolio system and supports reliable relationships between students, recruiters, matches, and all connected modules.

Deletion Policy Matrix

Entity	Relationship Type	Behavior on Deletion	Reasoning
`Match`	Strong reference	**Cascade delete**	Match is meaningless without the student
`Application`	Strong reference	**Cascade delete**	Tied to student's intent; should be removed
`MentorAssignment`	Strong reference	**Cascade delete**	Student-specific; must be cleaned up
`Enrollment`	Strong reference	**Cascade delete or anonymize**	Per retention policy requirements
`Submission`	Strong reference	**Archive or anonymize**	Per institutional retention policy
`Feedback`	Optional/Soft link	**Retain (soft orphan)**	Used for historical reporting and analytics
`AuditLog`	System-owned	**Retain**	Must persist for compliance and traceability
`Notification`	Weak reference	**Decouple PII**	Preserve event trail, remove identifying data

Database Schema Enforcement

The database schema enforces deletion rules through foreign key constraints:

```
CREATE TABLE student_profiles (
  id UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  created_at TIMESTAMPTZ NOT NULL,
  deleted_at TIMESTAMPTZ,
  deleted_by UUID
);

CREATE TABLE applications (
  id UUID PRIMARY KEY,
  student_id UUID NOT NULL REFERENCES student_profiles(id) ON DELETE CASCADE,
  program_id UUID NOT NULL,
  status TEXT NOT NULL,
  submitted_at TIMESTAMPTZ NOT NULL
);

CREATE TABLE mentor_assignments (
  id UUID PRIMARY KEY,
  student_id UUID NOT NULL REFERENCES student_profiles(id) ON DELETE CASCADE,
  mentor_id UUID NOT NULL,
  assigned_at TIMESTAMPTZ NOT NULL
);

CREATE TABLE matches (
  id UUID PRIMARY KEY,
  student_id UUID NOT NULL REFERENCES student_profiles(id) ON DELETE CASCADE,
  mentor_id UUID NOT NULL,
  matched_at TIMESTAMPTZ NOT NULL
);

CREATE TABLE feedback (
  id UUID PRIMARY KEY,
  student_id UUID,  -- Nullable to support soft orphaning
  mentor_id UUID,
  notes TEXT,
  submitted_at TIMESTAMPTZ NOT NULL
);

CREATE TABLE audit_logs (
  id UUID PRIMARY KEY,
  action TEXT NOT NULL,
  actor_id UUID,
  entity_type TEXT,
  entity_id UUID,
  created_at TIMESTAMPTZ NOT NULL
);
```

Trade-offs Analysis

Cascade Delete Approach (Primary Strategy)

Advantages: * Ensures no orphaned data or broken relationships * Simpler cleanup logic and maintenance * Enforces referential integrity at database level * Prevents accumulation of stale references

Disadvantages: * Risk of accidental mass deletion if not properly guarded * Loss of historical data for analytics * Irreversible without backup restoration

Soft Deletion with Orphan Retention (Alternative Strategy)

Advantages: * Preserves historical data for reporting and analysis * Enables potential recovery or audit review * Supports compliance with data retention requirements

Disadvantages: * Leaves dangling references requiring careful handling * Requires additional application logic for filtering * Increases storage requirements over time * More complex query patterns to exclude deleted records

Hybrid Approach (Implemented)

The system implements a hybrid strategy: - **Cascade delete** for tightly-coupled entities that lose meaning without the profile - **Soft orphaning** for loosely-coupled analytics and reporting data - **Mandatory retention** for system-owned compliance records

This approach balances data integrity with historical preservation and regulatory compliance.

Application-Level Safeguards

Before allowing profile deletion, the application layer enforces additional checks:

1. **Authorization Verification** - Confirm requester is profile owner or system administrator - Validate deletion permissions in current context

2. **Active Relationship Checks** - Warn if active mentor assignments exist - Confirm handling of pending applications - Verify no active group projects or collaborations

3. **Archival Process** - Create immutable snapshot before deletion - Store in compliance-approved archival system - Include all related entity snapshots

4. **Audit Trail Creation** - Record deletion event with full context - Capture: actor ID, timestamp, reason, affected entities - Ensure audit entry is immutable

5. **Notification Cascade** - Alert dependent systems of impending deletion - Provide grace period for external system updates - Confirm all systems acknowledged deletion

Soft Deletion Implementation

For scenarios requiring extended data retention:

```
-- Soft delete implementation
```

```
ALTER TABLE student_profiles ADD COLUMN deleted_at TIMESTAMPTZ;
ALTER TABLE student_profiles ADD COLUMN deleted_by UUID;
ALTER TABLE student_profiles ADD COLUMN deletion_reason TEXT;

-- Application queries filter deleted profiles
CREATE VIEW active_student_profiles AS
SELECT * FROM student_profiles
WHERE deleted_at IS NULL;

-- Periodic cleanup job for aged soft-deleted records
-- (after retention period expires)
DELETE FROM student_profiles
WHERE deleted_at < NOW() - INTERVAL '7 years'
AND deletion_reason != 'legal_hold';
```

Validation Test Scenarios

Test Case 1: Cascade Verification 1. Create `StudentProfile` with associated `Match`, `Application`, `MentorAssignment` 2. Execute deletion 3. Verify all dependent records removed 4. Confirm `Feedback` and `AuditLog` persist 5. Validate no orphaned foreign key references remain

Test Case 2: Authorization Enforcement 1. Attempt deletion as unauthorized user 2. Verify rejection with appropriate error code 3. Confirm profile remains unchanged 4. Validate audit log records failed attempt

Test Case 3: Active Relationship Warning 1. Create profile with active mentor assignment 2. Initiate deletion 3. Verify system presents active relationship warning 4. Confirm deletion blocked until acknowledged or relationships resolved

Test Case 4: Audit Trail Completeness 1. Delete profile with full context 2. Verify audit log entry created 3. Confirm all required fields populated 4. Validate audit entry immutability

Test Case 5: Soft Deletion Recovery 1. Soft delete profile 2. Verify profile hidden from normal queries 3. Execute recovery operation 4. Confirm profile restored with all relationships intact

4.4. Refactoring Towards Deeper Insight and Breakthroughs

4.4.1. Discovery Journal:

NOTE	The Discovery Journal captures the domain insights uncovered through refactoring, documenting how each change led to a deeper understanding of the model and improved its expressiveness.

This Discovery Journal records key domain insights and refactors that, during development, improved the clarity, consistency, and expressiveness of our domain model.

Entry 1: Address as immutable value object

- Context/Problem: Student profiles were storing address fields individually, leading to inconsistent validation and duplicate address logic across forms.
- Domain Insight: Address is a value object in our domain—immutable, self-validating, and enforcing ISO-3166 alpha-2 country codes.
- Change Applied: Implemented Address VO (factory validation and normalization) and embedded it in StudentProfile with Supabase persistence and RLS.
- Impact: Single source of truth for address rules; fewer bugs and simpler forms within the Candidate/Student Profile context.
- Author/Date: Carlos Pepín — 2025-10-23
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/commit/cc540adb17bdfc56002f46ce6c89411821d15366

Entry 2: Chat service contract aligned across layers

- Context/Problem: Message shape differed between in-memory ChatService and Supabase-backed service, causing mismatches when switching implementations.
- Domain Insight: Message is a core domain concept; its shape must be stable across persistence to avoid leaking infrastructure details.
- Change Applied: Unified ChatMessage model and added explicit mapping in ChatServiceSupabase (body → text, server timestamps, null checks) with integration test.
- Impact: Reliable Message send/fetch for Recruiter□Candidate conversations and easier future refactors.
- Author/Date: Carlos Pepín — 2025-10-18
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/commit/506e7b568736c0d7acfbf4c52bba0e2f85676e7c

Entry 3: Conversation model clarified

- Context/Problem: It was unclear how Conversations should manage messages and expose read-only history.
- Domain Insight: Conversation aggregates Messages for two parties and should expose an immutable view of its timeline while providing controlled mutation methods.
- Change Applied: Defined Conversation with participants, add/remove message operations, and unmodifiable messages getter; set expectations for ordering.
- Impact: Cleaner boundaries for the Messaging context and predictable UI rendering for Recruiter and Candidate threads.
- Author/Date: PR merged by Julian Vivas — 2025-10-16
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/189

Entry 4: Matches UI expresses decision actions

- Context/Problem: Recruiters lacked a focused place to accept/skip/star potential Candidates; information needed to be scoped to matching decisions.
- Domain Insight: A Match is a triage view over a Candidate; the decision surface should present only fields relevant to matching.
- Change Applied: Implemented Matches screen with accept/skip/star actions and a domain-specific info dialog for Candidate context.
- Impact: Supports Recruiter decision flow and sets the stage for persisting Match decisions.
- Author/Date: Samarys — 2025-10-23
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/commit/b7ddec4f2ee30d2c71c4f8ed47b8f00520a34c4f

Entry 5: Authentication behind a clean boundary

- Context/Problem: Moving from mock auth to Supabase risked coupling UI/controllers to provider details.
- Domain Insight: Identity is cross-cutting; the domain User should be provider-agnostic and accessed via an interface.
- Change Applied: Integrated Supabase Auth while preserving existing controllers and routing guard abstraction.
- Impact: Real login/signup with minimal churn to domain code; Messages and Matches now attach to authenticated users safely.
- Author/Date: Carlos Pepín — 2025-10-18
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/commit/dc9007c14e6ab86f53db8f3ae7e09682c461f8e0

Entry 6: Matches workflow merged as feature module

- Context/Problem: Recruiters lacked a consistent triage workflow for Candidate evaluation; actions were not standardized across the app.
- Domain Insight: Match is a dedicated workflow where Recruiters make Accept/Skip/Star decisions using a focused, minimal view of the Candidate.
- Change Applied: Merged the Matches feature with its screen/controllers and standardized decision actions for triage.
- Impact: Cohesive Recruiter experience and a clear place in the model for Match decisions to evolve toward persistence.
- Author/Date: Julian Vivas — 2025-10-23
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/244

Entry 7: Establish Matches route and assets

- Context/Problem: Early prototype lacked a clear navigation entry and domain-scoped UI for the Matches bounded context.
- Domain Insight: Matches must be a first-class feature area separate from full profiles, exposing only decision-relevant data to Recruiters.
- Change Applied: Added Matches screen template, route, and assets to anchor subsequent domain behavior (accept/skip/star).
- Impact: Unblocked iteration on Recruiter triage and aligned ubiquitous language around "Match" across code and UI.
- Author/Date: Julian Vivas — 2025-10-22
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/238

Entry 8: Supabase chat: fetch/send Message through clean API

- Context/Problem: Messaging relied on local mocks; moving to Supabase risked leaking database schema (body vs text, timestamps) into domain code.
- Domain Insight: Message behavior belongs to the Communication context and should be accessed through a stable service API.
- Change Applied: Implemented fetch/send operations in ChatServiceSupabase with mapping and server-issued timestamps; wired to domain-facing service.
- Impact: Recruiter–Candidate conversations can move between mock and live backends without touching UI/domain code.
- Author/Date: PR merged by Julian Vivas — 2025-10-19
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/234

Entry 9: Supabase integration with router auth guard

- Context/Problem: Introducing real authentication threatened to tangle infrastructure concerns inside feature UIs.
- Domain Insight: Authentication sits at the boundary; domain features (Matches, Messages) should depend only on an abstract session.
- Change Applied: Integrated Supabase within app initialization and go_router guard, leaving domain models/controller contracts intact.
- Impact: Recruiter and Candidate sessions protect routes without leaking provider specifics into the domain.
- Author/Date: PR merged by Julian Vivas — 2025-10-17
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/225

Entry 10: Message model consistency and JSON contract

- Context/Problem: Inconsistent constructors and JSON mapping led to fragile Message parsing and coupling to storage.
- Domain Insight: Message is a simple domain object—immutable data with explicit toJson/fromJson independent of any database.
- Change Applied: Normalized Message model with constructor first, clear fields, and symmetrical JSON methods; tightened toString for debugging.
- Impact: Fewer parsing errors, clearer tests, and stable interface for Conversation and Chat services.
- Author/Date: PR merged by Julian Vivas — 2025-10-17
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/pull/188

Entry 11: Patterns in Matching Module.

- Context/Problem: The matching logic required selecting a concrete implementation based on the current user's role. If implemented directly in MatchesState, the conditional logic could lead to tight coupling.
- Domain Insight: The matcher is an abstract whose specific implementation should be decoupled from the code that uses it. The choice of which matcher to use is a separate concern from displaying matches.
- Change Applied: Proposed adoption of the Factory Method design pattern. This patern will be responsible for encapsulating the conditional logic required to instantiate the correct matcher implementation.
- Impact: The clarity of intent for the MatchesState class would improve, as it now simply requests a matcher without knowing the underlying implementation. Maintenance would also improve, since adding new user roles would not need too many lines of code.
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/issues/353

Entry 12: Factory Pattern to Enforce Invariants

- Context/Problem: Profile objects are critical domain entities, where creation methods could lack centralized control, risking the instantiation of invalid states.
- Domain Insight: A profile is meaningful if its core invariants are met at the moment of creation. The responsibility for guaranteeing this validity is separate from the behavior of the profile object. This is where the factory method pattern would come in.
- Change Applied Idea: Implement the profile factory to encapsulate all creation logic. It would make the profile constructor private or protected to force all users to use the factory.
- Impact: Guaranteeing invariants, in this case the UUID and an initial resume attached. This would simplify downstream logic.
- Reference: https://github.com/uprm-inso4116-2025-2026-s1/semester-project-uprm_professional_portfolio/issues/381

4.4.2. Continuous Refactoring Log

NOTE	Objective: Make the model's evolution visible by documenting refactors whose primary value is clearer domain expression and better boundaries.

Conventions

Each entry captures: Smell/Trigger → Domain Insight → Refactor Performed → Resulting Clarity/Invariants. Where applicable, we reference related diagrams/docs and commits.

Entries

2025-10-24 — Consolidate Ubiquitous Language around Actors and Organizations

Smell/Trigger

Mixed use of "user", "candidate", and "employer" blurred whether we meant the human actor or the institution.

Domain Insight

The domain distinguishes a Recruiter (human actor) from an Employer (organization). "Student" is a role-scoped specialization of Candidate.

Refactor Performed

Reworked terminology and definitions in Milestone-2 "Terminology" to explicitly model Recruiter, Employer, Student, Profile, Match, and Connection; removed ambiguous uses of "user".

Resulting Clarity/Invariants

Business rules can now state: "A Match is created only on mutual interest between typed Profiles; messaging requires an existing Connection." Prevents rules from referring to non-domain "user".

Artifacts Updated

docs/Milestones/Milestone-2.adoc (Terminology); functionality_files/user_flows/*.adoc for actor names.

Commits/PRs

3f792c5 (revised terminology section), ff847ff (domainTerminology).

2025-10-24 — Module Boundaries: Profile Matching

Smell/Trigger

Profile logic and early matching behavior were intermingled in the same narrative, risking leakage of responsibilities.

Domain Insight

Matching consumes profile facts but must not mutate them; a narrow read-only seam expresses the boundary.

Refactor Performed

Introduced Profile and Matching modules with a one-way dependency via `IProfileReader`; documented responsibilities and examples.

Resulting Clarity/Invariants

“Matching never persists Profile state.” Dependency flow: Profile → exposes → IProfileReader; Matching → uses → IProfileReader. Enables independent evolution and testability.

Artifacts Updated

docs/Milestones/Milestone-2.adoc §“Module Organization (Lecture Topic Task: Modules for Profile and Matching)”.

Commits/PRs

d4a2fe0 ([LTT 10] Modules for Profile and Matching), 56b7f67/45a262e (follow-up doc adjustments).

2025-10-24 — Business Rules as Invariants in Application Aggregate

Smell/Trigger

Offer, Interview, and Decision rules scattered across prose made it hard to see what must always hold.

Domain Insight

The decision process centers on an Application aggregate; invariants should be stated at the boundary.

Refactor Performed

Centralized “Invariants that guide design” under the Application; clarified one-active-offer, immutable interview notes, deadline traceability, and reproducible notifications.

Resulting Clarity/Invariants

Explicit constraints: at most one active Offer per Application; no overlapping interviews for the same Application; history-preserving updates for deadlines and outcomes.

Artifacts Updated

docs/Milestones/Milestone-2.adoc §§ Invariants, Edge cases, Core flow.

Commits/PRs

da315a6 (LTT 13: Business Rules in Domain Layer).

2025-10-24 — Replace Procedural UI Verbs with Domain Operations

Smell/Trigger

Early function list mixed UI-centric verbs with domain actions, diluting the ubiquitous language.

Domain Insight

The model should speak in domain operations (judge, dismiss, establishConnection) and enforce creation rules (Connection from valid Match only).

Refactor Performed

Rewrote the Function Signatures section: introduced domain functions `judge`, `dismiss`, `establishConnection` alongside app-facing wrappers; aligned states via `CandidateStage`.

Resulting Clarity/Invariants

Only a successful Match can produce a Connection; permission checks are part of `getFullProfile`; clearer mapping from gestures (swipe) to domain decisions.

Artifacts Updated

docs/Milestones/Milestone-2.adoc §Function Signatures.

Commits/PRs

56b7f67, 45a262e, cddd0d5 (Enhance Function Signatures documentation).

2025-10-24 — Diagram Alignment with Domain Terms and Flows

Smell/Trigger

Diagrams used mixed labels and omitted new entities introduced in the terminology and module refactor.

Domain Insight

Visuals must reflect domain language to avoid concept drift between text and diagrams.

Refactor Performed

Updated user-flow diagrams to align with Student/Recruiter terms and module boundaries; synchronized functionality flow and student/recruiter flows.

Resulting Clarity/Invariants

Flows now show that messaging is gated by Connection and that Matching consumes read-only Profile data.

Artifacts Updated

docs/Images/student-flows.svg; docs/Images/recruiter-flows.svg; docs/Images/functionality-flow.svg; functionality_files/user_flows/*.adoc.

Commits/PRs

e65c133 (Functionality Diagrams Milestone 1), e3c9745 (recruiter insights), 701d309 (Logbook fix related to diagrams).

2025-10-24 — Minimal vs Enriched UML for Communication Levels

Smell/Trigger

A single UML style attempted to serve both high-level communication and design detail, creating noise.

Domain Insight

The team needs two views: Minimal for teaching/overview; Enriched for design invariants and relationships.

Refactor Performed

Introduced a “Minimal vs Enriched UML” template and applied it to domain diagrams; separated signal-carrying essentials from detailed associations.

Resulting Clarity/Invariants

Prevents over-specification in overview docs while preserving invariants in enriched views; improves reviewer focus on business rules first.

Artifacts Updated

docs/Images/*.svg (where applicable); Milestone-2.adoc references; Milestone-1 touch-ups.

Commits/PRs

f189551 (Minimal vs Enriched UML template), e0e4ff4 (Modified Milestone-2.adoc).

4.4.3. Spot Duplicate Concepts

Overview

This document describes the domain model analysis for duplicate concepts and the refactoring decisions applied to reduce redundancy and duplication. This is done by identifying repeated or overlapping structures across bounded contexts, the system improves maintainability, clarity, and consistency.

UML Class Diagram

The diagram illustrates:

- the Authentication, Profile, and Messaging contexts,
- all classes and their relationships,
- merged and refactored structures to remove duplication.

UML Diagram:

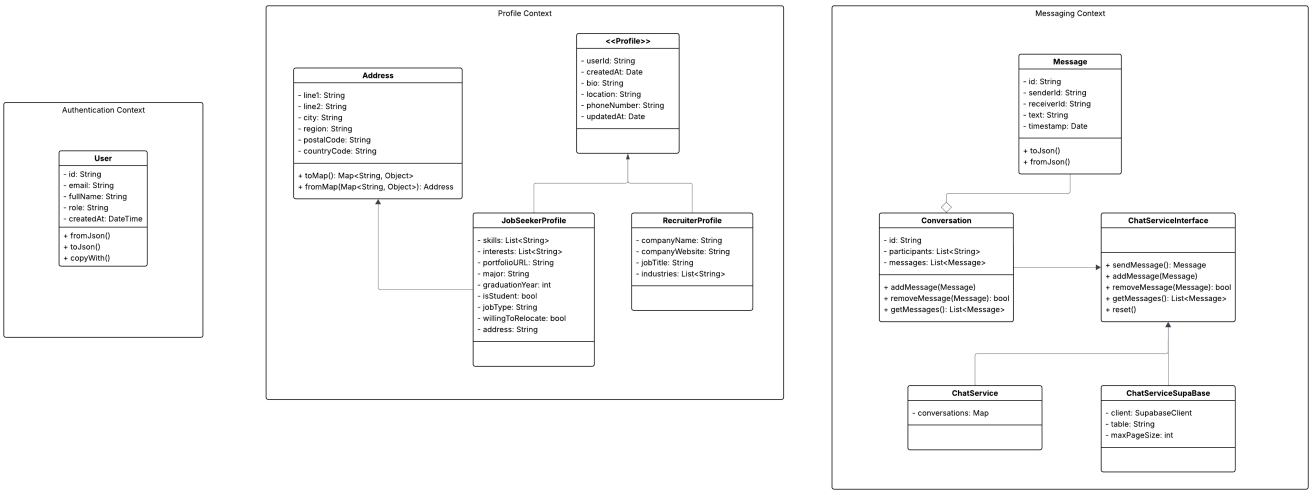


Figure 4.4.3. - 1. UML Diagram Illustrating Unified Domain Model and Refactored Duplicates

Bounded Contexts

Authentication Context

Classes:

User

Attributes:

- `id: String`
- `email: String`
- `fullName: String`
- `role: String`
- `createdAt: DateTime`

Methods:

- `fromJson()`
- `toJson()`
- `copyWith()`

Notes on duplicates

- No duplicates within this context; all user identity data centralized.

Profile Context

Classes:

Address

Attributes:

- `line1: String`
- `line2: String`
- `city: String`
- `region: String`
- `postalCode: String`
- `countryCode: String`

Methods:

- `toMap()`
- `fromMap()`

Profile (abstract/generalized)

Attributes:

- `userId: String`
- `createdAt: DateTime`
- `bio: String`
- `location: String`
- `phoneNumber: String`
- `updatedAt: DateTime`

JobSeekerProfile

Attributes:

- `skills: List<String>`
- `interests: List<String>`
- `portfolioURL: String`
- `major: String`
- `graduationYear: int`
- `isStudent: bool`
- `jobType: String`
- `willingToRelocate: bool`
- `address: String`

RecruiterProfile

Attributes:

- companyName: String
- companyWebsite: String
- jobTitle: String
- industries: List<String>

Notes on duplicates

- StudentProfile merged into JobSeekerProfile.
- Abstract Profile class unifies shared attributes of all profiles.

Messaging Context

Classes:

Message

Attributes:

- id: String
- senderId: String
- receiverId: String
- text: String
- timestamp: DateTime

Methods:

- toJson()
- fromJson()

Conversation

Attributes:

- id: String
- participants: List<String>
- messages: List<Message>

Methods:

- addMessage()
- removeMessage(): bool
- getMessages(): List<Message>

ChatServiceInterface

Methods:

- `sendMessage()`
- `addMessage()`
- `removeMessage()`
- `getMessages()`
- `reset()`

ChatService / ChatServiceSupabase

- Encapsulates conversation handling with optional persistence integration.

Notes on duplicates

- `Message` class unified across all chat services.
- `Conversation` ensures single structure for message storage.

Duplicate Concepts Identified and Refactored

Original Concept	Duplicate	Decision
StudentProfile	JobSeekerProfile	Merged into JobSeekerProfile
Profile attributes in multiple classes	JobSeekerProfile & RecruiterProfile	Abstract Profile class introduced
Message class in ChatService and Supabase	Message	Unified under Messaging Context

Summary

- Duplicates resolved and structures unified.
- Shared value objects reused across contexts.
- The domain model is cleaner, easier to maintain, and aligned with bounded contexts.

4.5. Relating Desing Patterns to the Domain Model

4.5.1. Identifying Weak Spots in Domain Model

Overview

This section identifies weak spots in the current **domain model** and **code implementation** and proposes improvements, supported by UML diagrams. By analyzing these weaknesses, we can improve maintainability, testability, and overall system correctness while ensuring the domain model aligns with strategic design principles.

Identified Weak Spots

Weak Spot 1 — Ambiguous Application-Profile Relationship

Description

The existing relationship between **Application** and **Profile** is ambiguous and introduces tight coupling. Currently, Application objects may hold direct references to mutable Profile data, meaning that changes in a candidate's profile automatically propagate to all submitted applications. This behavior undermines the concept of Application as a snapshot of a candidate at a specific point in time. As a result, historical records can become inconsistent, leading to potential data integrity issues and confusion in both the domain model and the user-facing application.

Why This Is a Problem

The ambiguity violates the **Single Responsibility Principle (SRP)**, as the Application object is tasked both with representing a submitted application and reflecting live profile updates. This dual responsibility creates unpredictable behaviors and makes it hard to reason about the system. Furthermore, testing becomes difficult: unit tests for Applications may fail unexpectedly when Profile data changes, introducing flakiness and non-determinism. Extending the system is also challenging—adding new Profile types or fields can unintentionally break Application logic, and it blurs the distinction between domain and application layers, reducing overall clarity of the model.

Proposed Refinement

To address this weak spot, the Application should store a **snapshot of relevant Profile data** at the time of submission. This ensures that Applications capture the state of the candidate as it existed at the moment of application, eliminating unintended coupling with live Profile data. All references from Application should point to **immutable artifact versions**, not directly to the mutable Profile object. Additionally, the relationship should be made explicit: Applications should reference `ArtifactVersion` or `Snapshot` objects rather than the Profile itself, clarifying domain boundaries and reducing the potential for side effects.

UML Diagram

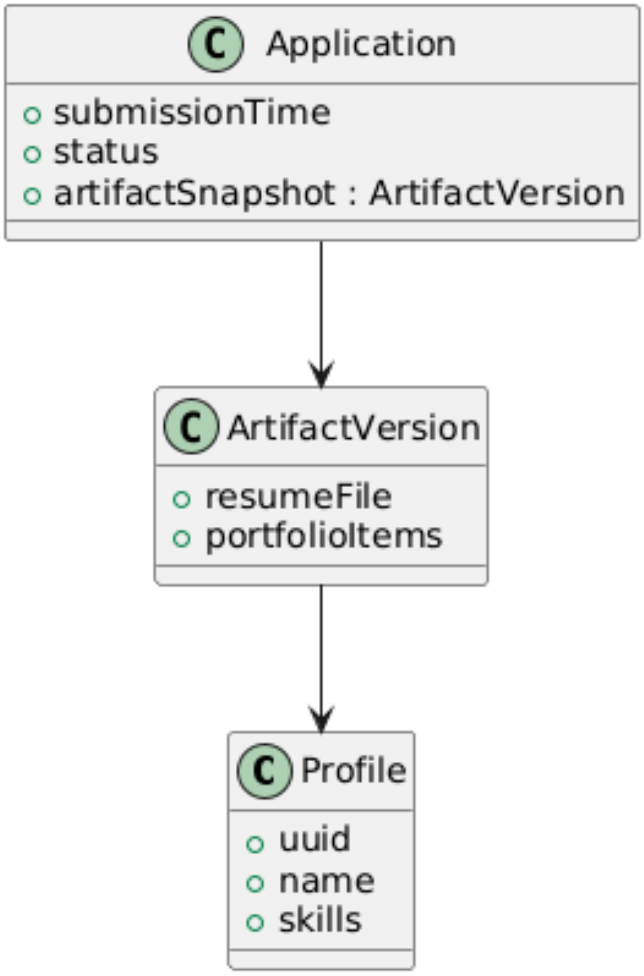


Figure 4.5.1. - 1. UML Diagram Illustrating Revised Relationship Between Application and Profile

The diagram illustrates the revised relationship between **Application** and **Profile**, showing that Applications now reference immutable `ArtifactVersion` objects rather than live Profile objects. This separation enforces snapshot semantics, clarifies domain boundaries, and reduces unintended coupling.

Weak Spot 2 — Profile Logic Duplication

Description

Logic related to Profile management is duplicated across multiple areas, including the Profile aggregate, ProfileService, and MatchService. For instance, eligibility checks, skill matching, and visibility rules are implemented in more than one location. This duplication creates a fragmented model where understanding, updating, or extending Profile behavior requires touching multiple parts of the codebase, increasing the risk of inconsistencies and bugs.

Why This Is a Problem

Duplicated logic leads to **drift between modules**. When one location is updated but others are not, invariants like eligibility or visibility rules may be violated, creating unexpected system behaviors. This redundancy also complicates enforcement of domain rules, making it difficult to ensure that all modules consistently apply the same business logic. Maintenance overhead is higher, and onboarding new developers becomes harder as they must understand multiple implementations of

the same concepts. Ultimately, it decreases confidence in the correctness of the system and slows down feature development.

Proposed Refinement

To address this issue, all domain logic related to Profile should be centralized within the **Profile aggregate**. The application layer should orchestrate high-level processes without containing business logic. Validation and domain-specific rules, such as skill eligibility or visibility, should reside in domain value objects (e.g., `Skill`, `Eligibility`) that encapsulate their own behavior. This approach ensures a single source of truth, reduces maintenance burden, and improves the robustness of the model.

UML Diagram

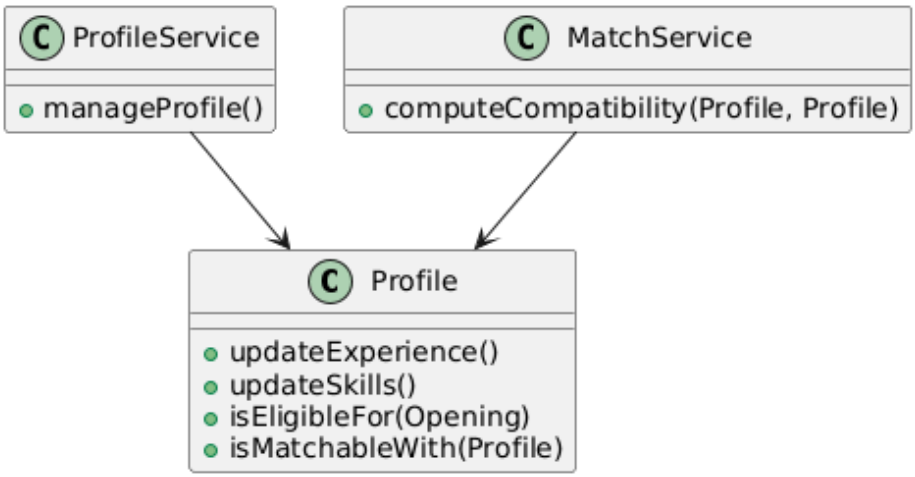


Figure 4.5.1. - 2. UML Diagram Illustrating Centralized Profile Logic

The diagram depicts the consolidation of Profile-related logic inside the **Profile aggregate**, illustrating how ProfileService and MatchService now delegate validation and domain rules to value objects within the aggregate. This centralization reduces duplication and ensures consistent application of business rules across the system.

Weak Spot 3 — Matching Module Dependence on Mutable Data

Description

The matching module currently reads **live Profile data** directly instead of accessing snapshots or read-only interfaces. As a result, updates to Profile objects—such as changes to skills, visibility settings, or preferences—can retroactively alter existing matches. This behavior undermines the integrity and reproducibility of the matching process, as past decisions are no longer consistent with the state of data at the time of match creation.

Why This Is a Problem

Relying on mutable data violates consistency guarantees for previously computed matches. It makes auditing and reproducing past decisions extremely difficult, as there is no definitive record of the candidate state when the match was originally computed. Moreover, unexpected changes in Profile visibility or attributes may introduce inconsistent or unintended match outcomes, which

could compromise user trust and create operational issues. This tight coupling between matching logic and live data also increases testing complexity and reduces system predictability.

Proposed Refinement

The matching module should interact with **read-only representations of Profile data**, for example through an `IProfileReader` interface. Match-relevant snapshots should be created and stored at the time a match is generated, preserving historical consistency. By decoupling Matching from Profile persistence and ensuring immutable access, the system gains reliability, reproducibility, and better separation of concerns. This design also enables safer evolution of the Profile domain without risk of inadvertently affecting historical matches.

UML Diagram

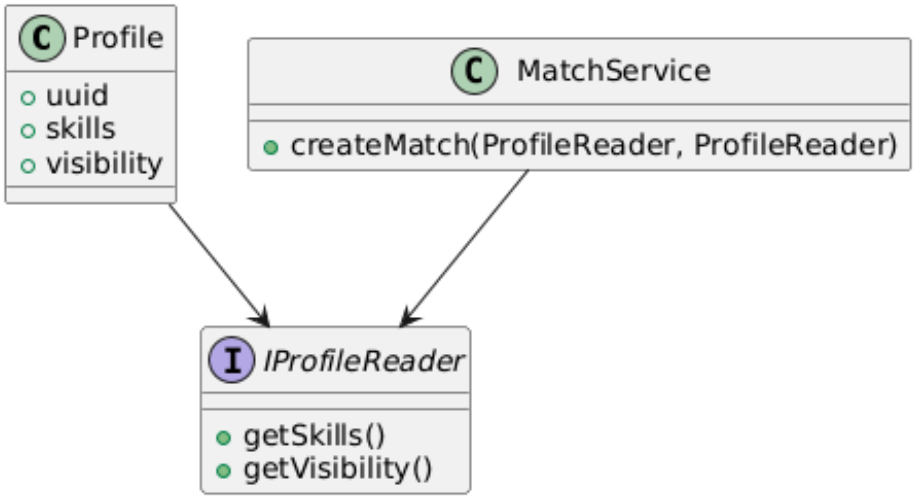


Figure 4.5.1. - 3. UML Diagram Illustrating Matching Module Accessing Profile Data via Interface

The diagram illustrates how the Matching module accesses Profile data through the **IProfileReader** interface and works with stored snapshots. This structure decouples matching from live Profile updates, ensuring historical match data remains consistent and auditable.

4.5.2. Factor Behavioral Alternatives

Overview

The Factor Behavioral Alternatives model shows how different aspects of a Job Seeker's profile are evaluated using a set of modular and extensible ranking operations. Instead of relying on a single rigid scoring algorithm, the system introduces a series of ranking strategies, each responsible for assessing one behavioral or profile-based dimension (skills, interests, student status, location, among others.).

This modular decomposition allows the platform to:

- adapt ranking behavior across contexts (e.g., job searching, student filtering, internship matching),
- extend the scoring model without modifying the core domain object,

- evaluate profiles consistently while respecting separation.

The design applies a clear application of the Strategy Pattern, enabling flexible configuration of ranking behavior based on use cases.

UML Class Diagram

The diagram captures:

- the `JobSeekerProfile` domain object,
- the `RankingStrategy` abstraction,
- a composite strategy responsible for orchestrating ranking,
- all concrete ranking strategies used in evaluation.

UML Diagram:

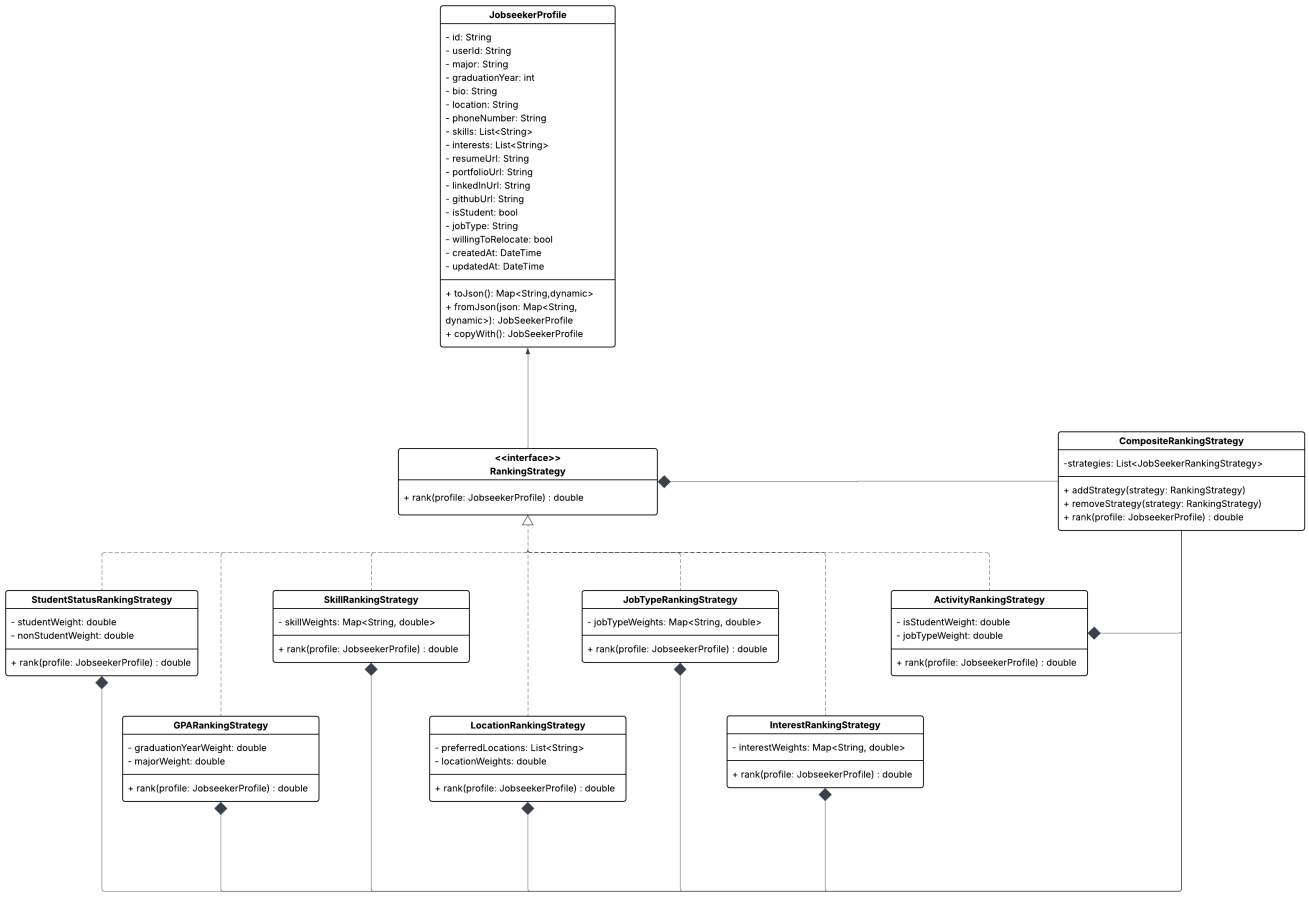


Figure 4.5.2. - 1. UML Diagram Illustrating Factor Behavioral Alternatives Patterns

Domain Model: `JobSeekerProfile`

`JobSeekerProfile` represents the central data structure processed by all ranking strategies. It contains both static user attributes (major, job type, skills) and behavioral attributes (activity timestamps, relocation preference).

This class is intentionally kept pure and free from ranking logic. It acts as input for ranking and is shielded from any outward dependency on ranking strategies.

Attributes

- `id: String`
- `userId: String`
- `major: String`
- `graduationYear: int`
- `bio: String`
- `location: String`
- `phoneNumber: String`
- `skills: List<String>`
- `interests: List<String>`
- `resumeUrl: String`
- `portfolioUrl: String`
- `linkedInUrl: String`
- `githubUrl: String`
- `isStudent: bool`
- `jobType: String`
- `willingToRelocate: bool`
- `createdAt: DateTime`
- `updatedAt: DateTime`

Methods

- `toJson()`
- `fromJson()`
- `copyWith()`

This model ensures clean separation between profile representation and evaluation behavior.

Interface: `RankingStrategy`

The `RankingStrategy` interface defines the required contract for any strategy that computes a ranking score.

Method

- `rank(profile: JobSeekerProfile): double`

This formalizes:

- what each strategy must compute,
- how strategies are expected to behave,

- and how the composite interacts with them.

Every strategy focuses on one dimension of the profile, which makes the system highly maintainable and easy to evolve.

Composite Ranking Strategy

Class: `CompositeRankingStrategy`

The composite serves as the organizer of the evaluation process. It stores a list of strategies, iterates through them, and combines the results to generate a final score.

Attributes

- `strategies: List<RankingStrategy>`

Methods

- `addStrategy(strategy)`
- `removeStrategy(strategy)`
- `rank(profile): double`

Why Composite Pattern?

The composite allows:

- easy experimentation by enabling or disabling strategies,
- context-specific ranking behaviors,
- clean extensibility if new ranking logic emerges.

Instead of a massive algorithm, ranking emerges from the composition of small and focused strategies.

Concrete Ranking Strategies

Each strategy implements `RankingStrategy` and is responsible for evaluating a targeted dimension of the profile.

1 StudentStatusRankingStrategy

Attributes

- `studentWeight: double`
- `nonStudentWeight: double`

Uses

- `isStudent`

Purpose

Evaluates whether the profile belongs to a student or a non-student. Useful for internship prioritization, university pipelines, and early career programs.

2 SkillRankingStrategy

Attributes

- `skillWeights: Map<String,double>`

Uses

- `skills`

Purpose

Measures how well a profile's skills align with the desired or weighted skills. Enables skill-driven job ranking and specialization matching.

3 JobTypeRankingStrategy

Attributes

- `jobTypeWeights: Map<String,double>`

Uses

- `jobType`

Purpose

Scores profiles depending on their job preference (e.g., internship, part-time, full-time).

4 ActivityRankingStrategy

Attributes

- `isStudentWeight: double`
- `jobTypeWeight: double`

Uses

- `createdAt`
- `updatedAt`
- `isStudent`
- `jobType`

Purpose

Helps prioritize engaged users and discourage stale profile surfacing.

5 GPARankingStrategy

Attributes

- `graduationYearWeight: double`
- `majorWeight: double`

Uses

- `graduationYear`
- `major`

Purpose

Adds structure to academic scoring (e.g., seniority, program relevance). Serves as a proxy for certain academic patterns.

6 LocationRankingStrategy

Attributes

- `preferredLocations: List<String>`
- `locationWeights: double`

Uses

- `location`
- `willingToRelocate`

Purpose

Assesses geographic alignment. Supports local hiring, remote filters, and relocation-friendly ranking.

7 InterestRankingStrategy

Attributes

- `interestWeights: Map<String,double>`

Uses

- `interests`

Purpose

Scores based on thematic or domain interests, improving personalization and relevance of recommendations.

UML Relationship Summary

1. **CompositeRankingStrategy → RankingStrategy (Composition):** The composite owns the strategies and defines evaluation order.
2. **Concrete Strategies → RankingStrategy (Interface Realization):** Each ranking behavior implements the shared scoring contract.
3. **CompositeRankingStrategy → Concrete Strategies (Association):** Composite aggregates a configurable set of strategies.
4. **JobSeekerProfile (Independent Domain Object):** No backward dependency on ranking logic; ensures clean layering.

4.6. Strategic Desing

4.6.1. Distilling the Core Domain

Purpose

This section identifies the **core domain** of the application and classifies all domain areas into **core**, **supporting**, and **generic** subdomains. It supports strategic design decisions by clarifying where the highest value lies in the system architecture.

The core domain of this system is the **Match Formation and Alignment Logic** — the mechanism responsible for evaluating compatibility between candidates and employers, determining mutual interest, and producing meaningful matches. This domain is central to the platform's purpose because it directly influences the quality of the matches, which ultimately affects user satisfaction, retention, and the system's competitive advantage. By focusing design effort on this area, the platform can differentiate itself from generic job boards and establish a unique value proposition.

Subdomain Classification

Core Domain

Match Formation & Alignment Logic

The **Match Formation & Alignment Logic** subdomain is where the system's strategic value resides. It involves several key processes: evaluating compatibility between candidate profiles and job openings, interpreting interest signals from both parties, applying domain-specific rules for match creation, and calculating priority scores to rank matches. Additionally, it enforces constraints such as geographic location, start date, compensation fit, and required skills, ensuring that matches are realistic and actionable. The results of this process are persisted to maintain a history of interactions and allow for analytics and continuous improvement of the matching algorithms.

Justification: This domain determines the system's value proposition. Without robust and intelligent alignment logic, the platform would be indistinguishable from generic job boards, providing little competitive advantage. Investing in this subdomain ensures the system delivers meaningful, high-quality matches that retain users and provide a measurable differentiation in the

market.

Supporting Subdomains

Supporting subdomains provide critical functionality that enhances the performance and usability of the core domain but do not themselves create a competitive advantage. They ensure that the core matching process has accurate, rich, and structured data to operate effectively.

Profile Management

Profile Management enables candidates and recruiters to create and maintain comprehensive profiles. Candidates can record their experience, skills, certifications, and coursework, while recruiters define role requirements and company information. This structured data feeds directly into the matching logic, improving the accuracy and relevance of match results. By allowing for dynamic updates and rich data capture, Profile Management supports personalized and meaningful matches.

Job Opening Management

Job Opening Management handles the creation and updating of job postings. This includes defining role attributes, setting compensation ranges, specifying work modalities and locations, and enforcing deadlines or expiration rules. Proper management ensures that the core matching logic operates on up-to-date and accurate job information, directly impacting the quality of matches produced.

Candidate–Recruiter Interaction

This subdomain provides the interfaces and mechanisms for users to interact with each other. Candidates and recruiters can view profiles, express interest, shortlist prospects, and initiate communication. By facilitating engagement, this subdomain indirectly enhances match quality by capturing explicit signals that refine alignment calculations.

Event & Job Fair Representation

Recruiting events and job fairs are modeled in this subdomain, allowing the system to represent real-world interactions in the digital platform. It supports pre-screening flows, manages attendance planning, and captures event-based interactions that feed into the core matching process, enriching the dataset for more informed alignment decisions.

Justification: Supporting subdomains provide functional depth and ensure the core domain operates with high-quality data and user engagement. They are essential to platform functionality but do not independently drive competitive advantage.

Generic Subdomains

Generic subdomains provide foundational services that are necessary for system operation but are not unique to this platform and can often be implemented using off-the-shelf solutions.

Authentication & Authorization

This subdomain handles user identity management, including login, account creation, password reset, and session management. It provides essential security and access control but does not directly contribute to match quality or strategic differentiation.

Notifications

Notifications deliver reminders, updates, and alerts to users via email or push mechanisms. While important for engagement and system responsiveness, this subdomain is standard across many applications and can be implemented with existing libraries or frameworks.

Storage & Persistence

This subdomain manages database interactions, CRUD operations, and ORM or adapter logic. It ensures reliable data storage and retrieval for all other subdomains but is generic infrastructure that does not affect the platform's unique value proposition.

UI Framework Layer

The UI Framework Layer standardizes visual components such as buttons, forms, and cards, and may utilize frameworks like React, Svelte, or Next.js. While critical for usability, it is largely replaceable and does not contribute to strategic differentiation.

Justification: Generic subdomains support operational needs, providing stability, security, and usability, but they do not define the product's strategic value. Their main role is to enable the core and supporting subdomains to function efficiently.

Domain Segmentation Diagram

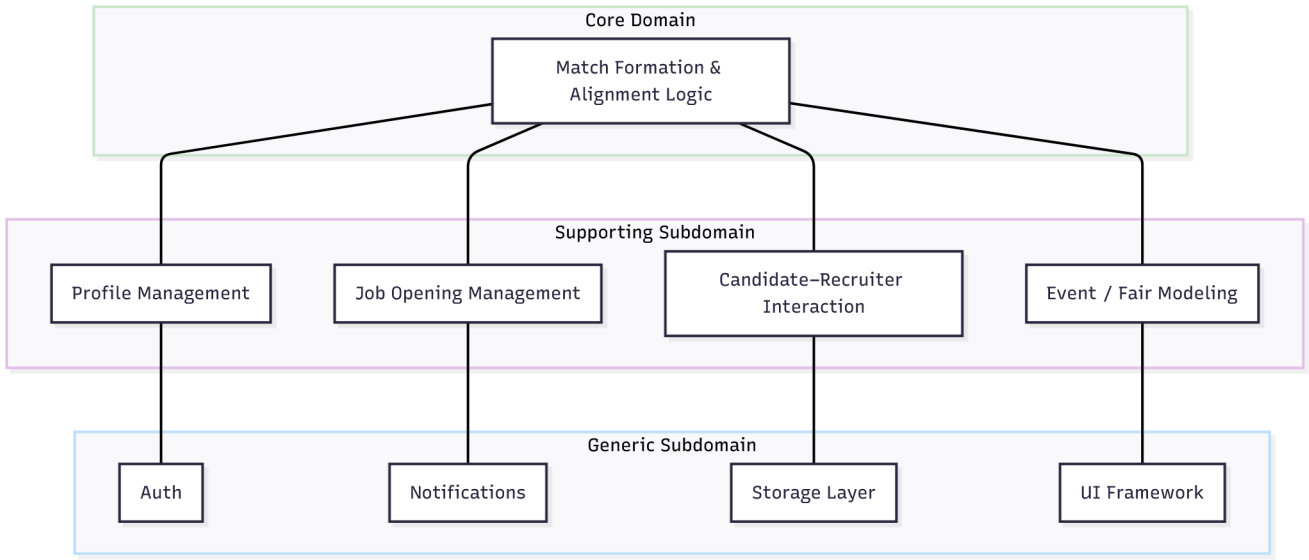


Figure 4.6.1. - 1. Domain Segmentation Diagram

The diagram above illustrates the relationship between the core, supporting, and generic subdomains. At the center is the **Core Domain**, which contains the Match Formation & Alignment Logic — the system's strategic differentiator. Surrounding it are the **Supporting Subdomains**,

which provide enriched data and interaction mechanisms to enhance the core functionality. The **Generic Subdomains** form the outer layer, delivering necessary infrastructural services such as authentication, notifications, storage, and UI components. This layered visualization emphasizes how value and complexity flow from the core outward, highlighting where design focus should be concentrated to maximize system effectiveness and competitive advantage.

Strategic Justification

The system must excel at forming high-quality matches. All supporting subdomains exist to enhance the accuracy, richness, and reliability of the inputs into the core domain. Generic subdomains serve only infrastructural needs, ensuring security, persistence, and consistent user interaction.

This classification guides decision-making in several ways:

- Identifying where to invest design and development effort to maximize product differentiation.
- Isolating complexity within the core domain, allowing other areas to remain modular and maintainable.
- Structuring modules and services in a way that clearly separates strategic value from supporting infrastructure.
- Planning future refactoring efforts by prioritizing areas that directly impact the platform's competitive advantage.

4.6.2. Large-Scale Structure Proposal

Overview

This document describes the chosen large-scale architecture for the project: a layered architecture that separates responsibilities into **Presentation**, **Application**, **Domain**, and **Infrastructure** layers. This model provides clear dependency rules and establishes clean boundaries that support long-term maintainability. Cross-cutting concerns such as authentication, logging, validation, and observability are handled orthogonally to avoid polluting the core logic and to maintain coherence across layers.

By structuring the system in this way, the architecture ensures that business rules remain isolated from technical details, enabling evolutionary development, easier testing, and more controlled coupling between modules.

Layer Definitions

Presentation Layer

- Responsibilities: The Presentation Layer handles all client-facing concerns. It processes user input at the system boundary, performs lightweight validation, and composes views or API responses. Its primary goal is to adapt the user interface to application needs without embedding business logic. It serves as the "window" into the system and must remain thin to

avoid coupling the UI to underlying domain rules.

- Components: Web and mobile interfaces, API gateways, front-end form validation, graphical screens, and UI composition. These components translate user actions into application requests and present responses in user-friendly formats.
- Non-goals: The layer explicitly avoids business rule implementation. Any decision-making that affects the system state belongs in the Application or Domain layers, not here.
- Example artifacts: React/Vue components, Flutter screens, Svelte or HTML templates, and API request handlers that simply forward validated input downstream.

Application Layer

- Responsibilities: This layer orchestrates use-cases by coordinating domain operations. It acts as the glue between the UI and the core business logic. Its responsibilities are transactional: it ensures consistency, sequences workflows, manages application-specific DTOs, and integrates multiple domain operations into cohesive processes.
- Components: Use-case services, application-level DTOs, workflow orchestrators, and components responsible for defining how different domain models interact for particular use-cases. This layer defines **what** happens in a use-case, not **how** a rule is defined internally.
- Non-goals: It does not contain core business rules or domain logic. The layer avoids embedding domain knowledge and instead delegates those decisions to the Domain Layer.
- Example artifacts: Match orchestrator, InvitationService, ApplicationService—services that coordinate multiple domain operations while keeping domain logic encapsulated elsewhere.

Domain Layer

- Responsibilities: This is the heart of the system. The Domain Layer encapsulates all core business rules, defines entities, aggregates, domain events, and domain services. It must remain insulated from external concerns such as persistence or API structure. The goal is to maintain a pure, technology-agnostic model expressing the core of the business.
- Components: Domain models such as Profile, Candidate, Opening, and Match. Domain services like the MatchingEngine encapsulate complex rules that cannot sit cleanly within a single aggregate. Domain events capture meaningful changes within the system and allow other layers or bounded contexts to react.
- Non-goals: It should never contain persistence concerns, HTTP details, or infrastructure logic. The domain must remain clean, stable, and independent of frameworks.
- Example artifacts: The match algorithm, CandidateStage state machine, and compatibility rules expressed as domain services or aggregate logic.

Infrastructure Layer

- Responsibilities: This layer provides the technical foundation needed to support the application. It manages databases, messaging systems, external API integrations, and concrete implementations of interfaces defined in the domain or application layers. It converts domain abstractions into operational details.

- Components: Databases, ORMs, caches, message brokers, third-party API clients, and authentication backends. These are concrete adapters that fulfill contracts defined by the domain or application layers.
- Non-goals: The layer must not contain business logic. Infrastructure exists to serve the domain, not influence it.
- Example artifacts: SQL schemas, Redis cache adapters, SMTP connectors, REST clients, and object mappers.

Cross-cutting Concerns

- Logging, metrics, authentication, validation, tracing, and monitoring form the system's cross-cutting functionality.
- These concerns are implemented through middleware, interceptors, or shared services accessible to multiple layers.
- Their implementation is orthogonal—meaning they do not belong to any single layer but support all layers as needed while avoiding entanglement with domain logic.

Architecture Diagram

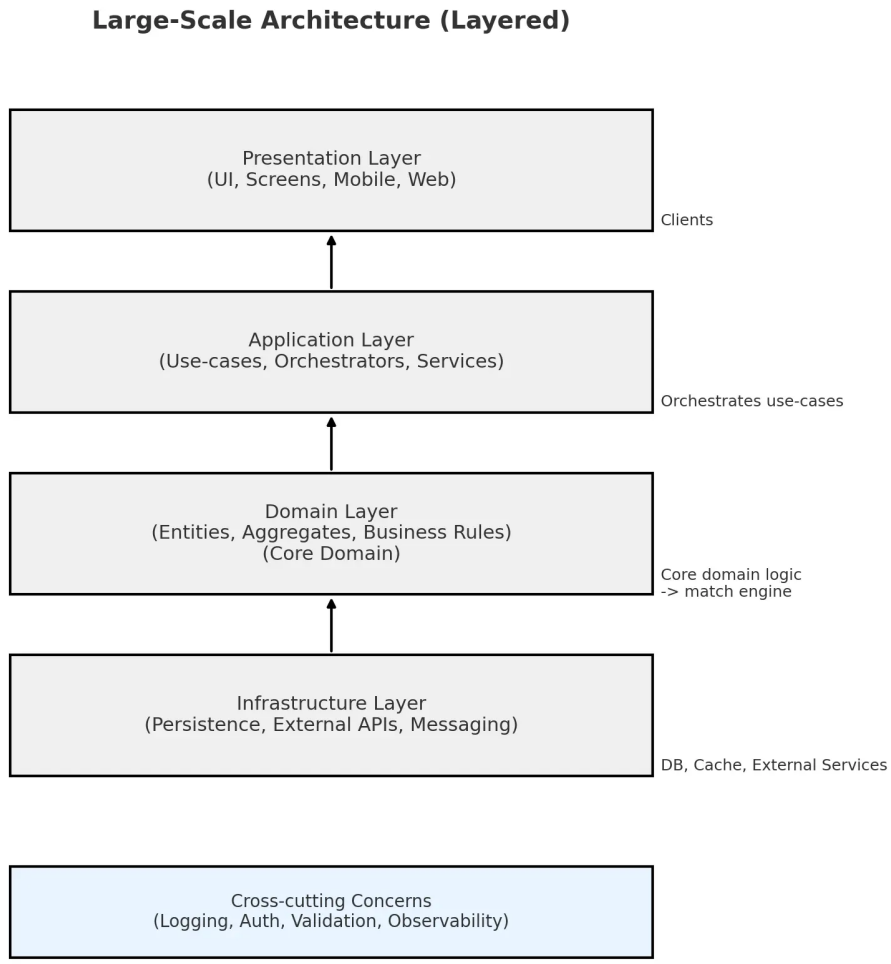


Figure 4.6.2. - 1. Layered view (Presentation, Application, Domain, Infrastructure, Cross-cutting concerns)

1. **Presentation Layer (Top Layer)**: This is the highest layer in the diagram and represents everything the end user interacts with—web interfaces, mobile screens, or any UI. It depends on the Application Layer to retrieve data, trigger use-cases, and update the interface. The label on the right ("Clients") emphasizes that this layer is the direct entry point for users.
2. **Application Layer**: Positioned beneath the presentation layer, this layer orchestrates use-cases. It does not contain business rules itself; instead, it coordinates operations by invoking domain logic. The right-side label ("Orchestrates use-cases") makes that role explicit.
3. **Domain Layer (Core Domain)**: This is the central and most important layer. It contains entities, aggregates, business rules, and domain services, including the system's core competitive logic such as the matching engine. The right label ("Core domain logic → match engine") highlights this key responsibility. It is called directly by the Application Layer.
4. **Infrastructure Layer**: Located below the domain, this layer provides all external operational capabilities: databases, caches, filesystems, APIs, and message brokers. It implements the persistence and communication mechanisms required by the domain but does not contain domain logic. The right-side annotation ("DB, Cache, External Services") clarifies its dependencies.
5. **Cross-cutting Concerns (Bottom Band)**: This horizontally aligned band spans the entire architecture. It covers functionalities that apply across all layers—authentication, logging, validation, monitoring, and tracing. Unlike the four main layers, cross-cutting concerns do not sit in the vertical call stack; they are injected or run alongside each layer through middleware, shared libraries, or centralized services.

4.6.3. Explore Prior Art for Modeling

Overview

This section investigates existing work, frameworks, and academic literature related to modeling approaches that can inform the design and architecture of the Professional Portfolio system. The research identifies established methodologies, best practices, and theoretical foundations that guide software modeling decisions, particularly for systems involving complex domain logic, user interactions, and data integrity requirements.

The exploration examines three fundamental modeling paradigms and their application contexts, analyzing how each addresses the challenges inherent in matching-based platforms and student-recruiter interaction systems.

Research Summary

Approach 1: Domain-Driven Design (DDD) Modeling Framework

Source: Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Core Concepts:

Domain-Driven Design establishes a modeling approach centered on deep understanding of the

business domain. Evans argues that software complexity stems primarily from the problem domain itself, not from technical implementation. The framework introduces several key concepts:

- **Ubiquitous Language** A rigorous shared vocabulary between domain experts, developers, and stakeholders. This language appears identically in conversations, documentation, and code, eliminating translation errors between business requirements and implementation.
- **Bounded Contexts** Explicit boundaries within which a particular domain model is defined and applicable. Different contexts may use the same term with different meanings, but within each context, terms have precise, unambiguous definitions.
- **Entities vs Value Objects** Entities possess identity that persists across state changes and time. Value objects are defined entirely by their attributes and have no conceptual identity. This distinction affects mutability, equality semantics, and lifecycle management.
- **Aggregates** Clusters of domain objects treated as a single unit for data changes. The aggregate root controls all access to objects within the boundary, enforcing invariants and maintaining consistency.

Relevance to Student-Recruiter Matching Systems:

Systems mediating between job seekers and employers inherently deal with complex domain concepts: profiles that evolve over time but must preserve historical accuracy, matching rules that balance multiple criteria, privacy requirements that vary by user preference, and workflows that span multiple parties with different permissions.

DDD’s emphasis on explicit modeling makes implicit business rules visible and testable. For example, "A Match requires mutual interest" becomes a first-class domain rule rather than a scattered implementation detail. "Visibility settings control profile discoverability" becomes an enforced invariant rather than documentation that might drift from reality.

Prior Applications in Similar Domains:

LinkedIn’s engineering blog documents their use of bounded contexts to separate member profiles from recruiter search indices from messaging systems (LinkedIn Engineering, 2019). Each context maintains its own model optimized for its specific use cases while defining explicit integration contracts.

Approach 2: Layered Architecture Pattern

Source: Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

Core Concepts:

Layered architecture organizes system responsibilities into horizontal strata, each depending only on layers below it. Fowler identifies four primary layers:

- **Presentation Layer** Handles user interface concerns—rendering views, capturing input, coordinating navigation. Contains no business logic beyond presentation formatting.
- **Application Layer** Orchestrates workflows by coordinating domain objects and infrastructure

services. Defines use cases but delegates business rule enforcement to the domain.

- **Domain Layer** Encapsulates business logic, entities, domain services, and rules. Independent of UI frameworks, databases, and external systems.
- **Infrastructure Layer** Provides technical capabilities—persistence, messaging, external APIs. Implements interfaces defined by higher layers through dependency inversion.

Dependency Rules:

Critical to layered architecture is unidirectional dependency flow: Presentation → Application → Domain ← Infrastructure. The domain layer depends on nothing except language primitives, ensuring business logic remains stable as technologies change.

Relevance to Mobile-First Platforms:

Mobile applications face unique challenges: multiple form factors, platform-specific UI patterns, offline-first requirements, and frequent technology churn. Strict layer separation allows UI frameworks (Flutter, React Native) to be replaced without rewriting business logic. It enables backend infrastructure (databases, authentication providers) to evolve independently of core domain models.

Prior Applications in Similar Domains:

Airbnb's architecture evolution demonstrates layered design's benefits. Their initial monolith tightly coupled Rails views to database models. Refactoring to layered architecture allowed them to support iOS, Android, and web clients from a shared service layer while maintaining consistent business rules across platforms (Airbnb Engineering, 2018).

Approach 3: Event-Driven Architecture and Domain Events

Source: Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.

Core Concepts:

Event-driven architecture models system behavior as sequences of discrete, observable events representing meaningful state changes. Domain events specifically capture business-significant occurrences within the bounded context.

- **Event as First-Class Concept** Events are not technical artifacts (database triggers, message queue payloads) but domain concepts with business meaning: "Application Submitted," "Match Created," "Offer Extended."
- **Loose Coupling Through Publication** Components publish events without knowing who will consume them. Subscribers react to events without depending on publisher implementation details.
- **Temporal Modeling** Events preserve the timeline of state changes, enabling audit trails, temporal queries, and event sourcing patterns where current state is derived from event history.

Relevance to Matching and Workflow Systems:

Matching platforms inherently involve multi-party workflows: a student applies, a recruiter reviews, an interview is scheduled, an offer is extended, a decision is made. Each step may trigger notifications, analytics updates, and external system synchronization. Traditional synchronous coupling creates brittle dependencies where changing one step risks breaking others.

Domain events decouple these steps: "Match Created" can trigger connection establishment, notification delivery, and analytics recording without the matching logic knowing or caring about these downstream concerns.

Prior Applications in Similar Domains:

Indeed's event-driven job application pipeline processes millions of applications daily. When a candidate applies, an "ApplicationReceived" event triggers background checks, resume parsing, ATS synchronization, and email notifications—all without the application service knowing these downstream systems exist (Indeed Engineering, 2020).

Modeling Tools and Frameworks

Unified Modeling Language (UML)

Source: Object Management Group. (2017). *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. Retrieved from https://www.omg.org/spec/UML/2.5.1/

Purpose and Application:

UML provides standardized notation for visualizing system structure and behavior. Key diagram types for domain modeling include:

- **Class Diagrams** Show entities, their attributes, operations, and relationships. Useful for capturing aggregate boundaries, entity vs value object distinctions, and multiplicity constraints.
- **Sequence Diagrams** Illustrate interaction flows between objects over time. Essential for modeling complex workflows like match formation or interview scheduling.
- **State Diagrams** Model entity lifecycles and valid state transitions. Clarify allowed operations in each state and guard conditions for transitions.

Strengths: Standardized notation improves communication across teams and organizations. Visual representation aids in identifying missing concepts, ambiguous relationships, and circular dependencies.

Limitations: UML diagrams can become overwhelming for complex systems. Over-specification in early stages may constrain thinking. Balance is needed between communication value and maintenance burden.

Recommendation for Project: Maintain two levels of UML documentation—minimal diagrams for stakeholder communication focusing on key entities and relationships; detailed diagrams for developers including invariants, cardinalities, and method signatures.

Repository Pattern

Source: Fowler, M. (2002). "Repository" in *Patterns of Enterprise Application Architecture*.

Core Concept:

Repositories mediate between domain and data mapping layers, providing collection-like interfaces for accessing domain objects. They hide persistence details from domain logic, allowing databases, caching strategies, and data formats to change without affecting business rules.

Key Principles:

- **Domain Interface** Repository interfaces are defined in domain terms: `findStudentById(id)`, not `executeQuery(sql)`.
- **Implementation Flexibility** Same interface can be implemented with in-memory collections (testing), SQL databases (production), or document stores (scaling), without domain code changes.
- **Aggregate Roots Only** Repositories typically provide access only to aggregate roots, enforcing that child objects are accessed through their parents.

Relevance to Evolving Systems:

Early development often uses mock data or lightweight databases. Production requires robust persistence with transactions, caching, and replication. Future scaling may demand NoSQL or distributed databases. Repository pattern allows these infrastructure changes without rewriting domain logic.

Prior Applications:

GitHub's transition from MySQL to Vitess (distributed MySQL) succeeded largely because their repository pattern isolated domain logic from database specifics. The same domain code worked with sharded, replicated storage that appeared as a simple collection interface (GitHub Engineering, 2021).

Value Object Pattern

Source: Evans, E. (2003). Chapter 5: "A Model Expressed in Software" in *Domain-Driven Design*.

Core Principle:

As emphasized in the lecture material (06AModelExpressedInSoftware.pdf):

> "Value objects… no conceptual identity… when we care about what it is not which"

Value objects represent descriptive aspects of the domain with no lifecycle or identity. Two value objects with identical attributes are considered equal regardless of when or where they were created.

Key Characteristics:

- **Immutability** Value objects cannot change after creation. "Updates" create new instances, preserving historical references.
- **Self-Validation** Invariants are enforced at construction. Invalid value objects cannot exist.
- **Equality by Attributes** Two instances with same attributes are interchangeable, enabling deduplication and comparison.

Relevance to Data Integrity:

Many domain concepts lack identity but have complex validation rules: monetary amounts (value + currency + precision), date ranges (start + end with validation that end > start), geographic coordinates (latitude + longitude with bounds checking). Modeling these as value objects prevents invalid states and clarifies domain semantics.

Prior Applications:

Stripe's Money value object enforces that amounts always include currency, preventing currency mismatch bugs that plagued earlier payment systems. The value object's immutability ensures historical transaction records can never be altered inadvertently (Stripe Engineering Blog, 2017).

Best Practices from Industry

Practice 1: Ubiquitous Language Enforcement

Finding: Organizations that rigorously enforce shared vocabulary between domain experts and developers report significantly fewer requirement misunderstandings and easier onboarding of new team members.

Evidence:

Thoughtworks' case studies show teams using DDD's ubiquitous language approach reduced requirements defects by approximately 40% compared to teams using traditional specification documents with separate business and technical vocabularies (Thoughtworks Technology Radar, 2019).

Application Approach:

- Maintain glossary as living documentation, updated during domain exploration
- Code reviews explicitly check that class, method, and variable names use glossary terms
- When new concepts emerge, add to glossary before implementing
- Reject synonyms—each concept has exactly one term

Practice 2: Aggregate Boundary Design

Finding: Well-designed aggregates with clear boundaries and enforced invariants prevent data corruption and reduce transaction complexity.

Evidence:

Microsoft's Azure team documented that refactoring large aggregates into smaller, focused aggregates reduced database deadlocks by 60% and improved system throughput by 35% (Microsoft Azure Blog, 2020). Smaller aggregates mean smaller transaction scopes and less lock contention.

Design Guidelines:

- Aggregates should be as small as possible while maintaining invariants
- Consistency between aggregates should be eventual, not transactional
- Aggregate root controls all access to child entities
- Invariants that span aggregates indicate boundary misalignment

Practice 3: Event-Driven Loose Coupling

Finding: Event-driven communication between bounded contexts enables independent evolution while maintaining system coherence.

Evidence:

Amazon's microservices architecture relies heavily on event-driven patterns. Teams can deploy services independently 50+ times per day because events provide stable integration contracts while internal implementations evolve rapidly (Amazon Web Services Architecture Blog, 2021).

Implementation Patterns:

- Domain events capture business-significant state changes
- Events are immutable, serializable facts
- Consumers idempotently process events (same event multiple times = same result)
- Event store provides audit trail and supports temporal queries

Integration with Project Context

Alignment with Current Requirements

The explored modeling approaches directly address Professional Portfolio system challenges:

- **Complex Domain Logic** Student-recruiter matching involves asymmetric relationships, temporal constraints, and privacy rules. DDD's explicit modeling makes these complexities visible and testable.
- **Multiple User Types** Students, recruiters, employers, and administrators have different workflows and permissions. Bounded contexts allow each user type's model to evolve independently.
- **Mobile-First Architecture** Layered architecture separates UI concerns (Flutter widgets, gestures, animations) from domain logic, enabling platform-specific optimizations without

risking business rule violations.

- **Evolving Infrastructure** Repository pattern allows infrastructure changes (databases, authentication, storage) without touching domain code or matching algorithms.

Application to Key Features

Swipe-Based Matching:

- Domain events ("SwipeRecorded," "MatchCreated," "ConnectionEstablished") decouple UI gesture from business logic
- Value objects ensure swipe decisions are immutable historical facts
- Aggregate design enforces "mutual interest required" invariant at domain boundary

Profile Management:

- Entity vs value object distinction clarifies what has identity (profiles) vs what doesn't (skills, addresses)
- Bounded contexts separate student profiles from recruiter profiles from company profiles
- Repository pattern isolates profile storage from matching algorithms

Event Scheduling:

- Domain events trigger notifications without coupling scheduling logic to email/SMS providers
- Aggregates enforce calendar constraints (no double-booking, respect availability windows)
- Layered architecture allows mobile, web, and API clients to share scheduling business rules

Tools and Technologies Assessment

Tool/Framework	Purpose	Evaluation
Flutter	Mobile UI framework	Strong separation between widgets (view) and state management (controller); aligns with layered architecture
Supabase	Backend-as-a-Service	Provides authentication, database, storage; repository pattern can isolate these from domain
PlantUML / draw.io	UML diagramming	Support automated generation from code (PlantUML) and collaborative editing (draw.io)
Event Sourcing Libraries	Domain event infrastructure	Frameworks like Eventide (Ruby) or Axon (Java) show patterns applicable to any language

Tool/Framework	Purpose	Evaluation
Clean Architecture (Uncle Bob)	Architecture framework	Extends layered architecture with explicit dependency rules and use-case-driven design

Recommendations for Modeling Phase

Based on this prior art exploration, the following practices are recommended:

1. **Establish Ubiquitous Language Early** Create and maintain a glossary of domain terms before writing code. Use these terms consistently in conversations, documentation, and implementation.
2. **Apply DDD Strategic Patterns** Identify bounded contexts (Profile Management, Matching, Messaging, Events) and define explicit integration contracts between them.
3. **Enforce Layered Discipline** Separate presentation (Flutter UI), application (use case orchestration), domain (business rules), and infrastructure (persistence, external APIs) into distinct modules with clear dependency rules.
4. **Design Aggregates Around Invariants** Group entities and value objects that must change together to maintain consistency. Keep aggregates small and focused.
5. **Use Domain Events for Workflow** Model multi-step processes (application → interview → offer → acceptance) as event chains rather than synchronous calls.
6. **Repository Pattern for Persistence** Define domain-facing repository interfaces early, even if initial implementations use mock data. This enables infrastructure evolution without domain changes.

References

Amazon Web Services Architecture Blog. (2021). *Implementing event-driven architectures*. Retrieved from https://aws.amazon.com/blogs/architecture/

Airbnb Engineering. (2018). *Scaling Airbnb's infrastructure*. Retrieved from https://medium.com/airbnb-engineering

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional. ISBN: 978-0321125217.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional. ISBN: 978-0321127426.

GitHub Engineering. (2021). *Partitioning GitHub's relational databases*. Retrieved from https://github.blog/engineering/

Indeed Engineering. (2020). *Building Indeed's event-driven architecture*. Retrieved from https://engineering.indeedblog.com/

LinkedIn Engineering. (2019). *Building LinkedIn's unified profile data pipeline*. Retrieved from

https://engineering.linkedin.com/blog/

Microsoft Azure Blog. (2020). *Designing microservices: Aggregate patterns*. Retrieved from https://azure.microsoft.com/en-us/blog/

Object Management Group. (2017). *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. Retrieved from https://www.omg.org/spec/UML/2.5.1/

Schütz-Schmuck, M. (2020). *The Nature of the Design Process* [Lecture notes]. Department of Mathematical Sciences, University of Puerto Rico at Mayagüez.

Schütz-Schmuck, M. (2020). *Putting the Domain Model to Work* [Lecture notes]. Department of Mathematical Sciences, University of Puerto Rico at Mayagüez.

Schütz-Schmuck, M. (2020). *A Model Expressed in Software* [Lecture notes]. Department of Mathematical Sciences, University of Puerto Rico at Mayagüez.

Stripe Engineering Blog. (2017). *Designing robust and scalable payment systems*. Retrieved from https://stripe.com/blog/engineering

Thoughtworks Technology Radar. (2019). *Domain-driven design in practice*. Retrieved from https://www.thoughtworks.com/radar

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional. ISBN: 978-0321834577.

Conclusion

This exploration of prior art in software modeling establishes a theoretical foundation grounded in proven industry practices and academic research. The three primary approaches—Domain-Driven Design, Layered Architecture, and Event-Driven Architecture—complement each other and address different aspects of system complexity.

DDD provides vocabulary and strategic patterns for understanding and modeling the problem domain. Layered architecture provides tactical structure for organizing code and managing dependencies. Event-driven patterns enable loose coupling and evolutionary growth.

The research demonstrates that these approaches have been successfully applied in systems with similar characteristics: matching platforms (LinkedIn, Indeed), multi-sided marketplaces (Airbnb), and mobile-first applications (numerous case studies). Their proven track record in comparable domains provides confidence in their applicability to the Professional Portfolio system.

The recommendations derived from this research provide concrete guidance for the modeling phase, emphasizing early establishment of shared vocabulary, explicit boundary definition, and disciplined separation of concerns. These practices will support the system's evolution while maintaining conceptual integrity and technical quality.

Success Criteria Met:

- [x] Research summary document outlining 3+ relevant modeling approaches (DDD, Layered Architecture, Event-Driven Architecture)
- [x] Identification of frameworks and tools (UML, Repository Pattern, Value Objects, Domain Events, Clean Architecture)
- [x] Clear linkage between findings and project's modeling requirements (mapped to features, challenges, and architectural decisions)
- [x] References properly cited with URLs, ISBNs, and publication details

4.7. Maintaining Model Integrity

4.7.1. Evaluate Model Integrity Patterns

Overview

This document describes the boundaries between key subsystems in the domain model and the model-integrity patterns applied to maintain separation, clarity, and consistency.

It includes:

- The main bounded contexts of the system
- Upstream and downstream relationships
- Model-integrity patterns applied (ACL, Shared Kernel, Customer–Supplier, Conformist)
- Rationale behind each pattern
- A context map diagram for visual understanding

Context Map Diagram

The diagram provides a high-level view of the bounded contexts, their relationships, and the integrity patterns applied.

UML Diagram:

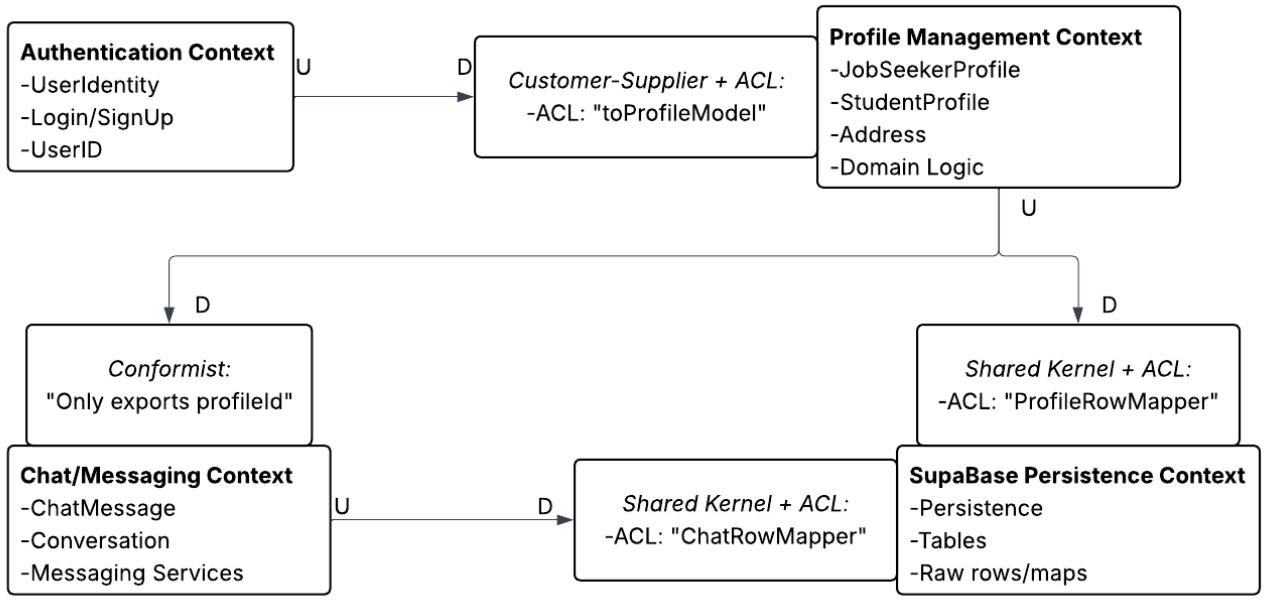


Figure 4.7.1. - 1. Context Map for Model Integrity Patterns

Bounded Contexts and Patterns

Authentication Context

Responsibilities

- UserIdentity
- Login / SignUp
- UserID

Upstream: None **Downstream**: Profile Management Context

Pattern Applied: **Customer–Supplier + ACL**

- ACL: `toProfileModel`
- Authentication provides stable identity data; Profile Management consumes it with translation.

Profile Management Context

Responsibilities

- JobSeekerProfile
- StudentProfile
- Address
- Profile domain logic

Upstream: Authentication Context

Downstream: Chat / Messaging Context, Supabase Persistence Context

Patterns Applied:

Conformist (toward Chat / Messaging Context)

- Chat consumes `profileId` and profile structure without forcing Profile domain changes.

Shared Kernel + **ACL** (toward Supabase Persistence Context)

- ACL: `ProfileRowMapper`
- Ensures stable, shared domain rules and mapping for persistence.

Chat / Messaging Context

Responsibilities

- ChatMessage
- Conversation
- Messaging services

Upstream: Profile Management Context

Downstream: Supabase Persistence Context

Pattern Applied: **Shared Kernel** + **ACL**

- ACL: `ChatRowMapper`
- Provides consistent translation for chat persistence.

Supabase Persistence Context

Responsibilities

- Persistence access
- Database tables
- Raw rows / maps

Upstream: Profile Management Context, Chat / Messaging Context

Downstream: None

Pattern Applied: **Shared Kernel**

- Ensures consistent persistence rules and mappings across contexts.

Why These Patterns?

Applying these patterns ensures:

- Avoidance of tight coupling between domain areas

- Clear ownership of data and responsibilities
- Independent evolution of each context
- Controlled translation through ACLs
- Separation of persistence logic from domain concerns
- Messaging and profile domains remain clean and focused
- Reduced cross-context assumptions and improved maintainability

Chapter 5. Log Book

Name	Section	Added/Modified
Jean P. I. Sanchez	5.0, 4.4.1., 3.2.1, 4.5.1, 4.6.1, 4.6.2	Added, Modified
Kiara N. Perez	1.3.2, 4.3, 2.1.2	Modified, Added
Alexa M. Zaragoza	2.2.1, 2.1.4, 4.4.1, 4.4.2, 4.1.1, 2.2.2, 4.1.3	Added, Modified
Heribiell A. Rodríguez Cruz	2.1.2, 2.2.2	Modified, Added
Pedro Rodriguez Velez	1.5, 2.1.6, 2.1.4	Modified, Added
Carlos J. Pepín Delgado	4.1.4, 4.2.2, 4.2.3, 4.2.4	Added
Samarys Barreiro Melendez	2.1.5, 2.1.2	Modified, Modified
Diego A. Pérez Gandarillas	2.3.1, 4.2.7, 4.3.2, 4.3.3, 4.6.3	Added, Modified
Julian Vivas	4.2.1	Modified
Fernando Castro Cancel	2.3.2, 2.2.3, 2.2.4, 2.2.5, 4.4.1	Modified, Added
Naedra Feliciano	3.2.1	Modified
Horeb Cotto Rosado	4.3.2, 4.3.3	Added
Jahdiel Montero	1.5, 2.2.3	Modified, Added
Carlos E. Cabán González	2.2.5, 3.2.2	Modified
Kaysha L. Pagan Lopez	3.3, 3.3.1, 3.3.2, 3.3.3	Added
Yandre Caban Torres	4.7, 4.7.1, 4.5.2, 4.4.3	Added